AFIT/GCS/ENG/96J-01

A METHODOLOGY FOR REENGINEERING RELATIONAL DATABASES TO AN OBJECT-ORIENTED DATABASE

THESIS

Pedro A. Linhares Lima, Major, Brazilian Air Force

AFIT/GCS/ENG/96J-01

Approved for public release; distribution unlimited

AFIT/GCS/ENG/96J-01

A METHODOLOGY FOR REENGINEERING RELATIONAL DATABASES TO AN OBJECT-ORIENTED DATABASE

THESIS

Presented to the faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Science

Pedro A. Linhares Lima, B. S.

Major, Brazilian Air Force

JUNE, 1996

Approved for public release; distribution unlimited.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

Acknowledgments

I would like to thank my advisor, Doctor Thomas C. Hartrum, for his trust, guidance, and assistance during this research effort. He provided motivation without limiting the learning from the freedom to explore. I also wish to thank my readers, Doctor Henry Potoczny and Doctor Eugene Santos Jr.

Special thanks to Doctor Blaha for explaining some points of his methodology and answering some questions concerning reengineering tools.

Finally, I would like to thank my wife, Ana Cristina R. Linhares Lima, for her support and understanding during the time consuming thesis process. She provided encouragement and motivation to complete the thesis and the AFIT program.

Pedro Arthur Linhares Lima

Table of Contents

ACKNOWLEDGMENTS IV

TABLE OF CONTENTS V

TABLE OF FIGURES VIII

ABSTRACT X

1 Introduction 1

1.1 Background. 1

1.2 Problem. 3

1.3 Hypothesis. 3

1.4 Research Objectives. 3

1.5 Test case. 4

1.6 Assumptions. 5

1.7 Sequence of Presentation. 6

2 Summary of Current Knowledge 7

2.1 Treatment and Organization. 7

2.2 Software Reengineering. 7

2.3 Reengineering of Relational Databases. 9

2.4 Object-Oriented Methodology. 14

2.5 Conclusion 15

3 Methodology 16

3.1 Introduction 16

3.2 The Methodology 16

3.3 Summary 27

4 Application of Methodology **28**

4.1 Introduction 28

4.2 ER Model 28

4.3 Object Model 31

4.4 Functional Model 36

4.5 Summary 37

5 Implementation Issues **38**

5.1 Introduction 38

5.2 Analysis and Choice of the OODBMS 38

5.3 Implementation of the Object Model 39

5.4 Limitations of Foxpro Encountered During Implementation 43

5.5 Summary 44

6 Analysis, Conclusions, and Recommendations **45**

6.1 Analysis of The Results 45

6.2 Conclusion 46

6.3 Recommendations 48

APPENDIX A: LIST OF TABLES WITH SSAN **50**

APPENDIX B: THE ENTITY RELATIONSHIP DIAGRAM **54**

APPENDIX C: THE OBJECT MODEL **58**

APPENDIX D: THE FUNCTIONAL MODEL 82

APPENDIX E: IMPLEMENTATION OF THE OBJECT MODEL 90

APPENDIX F: LIST OF ABBREVIATIONS 101

BIBLIOGRAPHY 102

VITA

Table of Figures

FIGURE 1: RELATIONSHIP BETWEEN TERMS [5] 8

FIGURE 2: VARIOUS APPROACHES TO IDENTIFY THE PRIMARY KEY [2] 19

FIGURE 3: DOUBLE BURIED ASSOCIATION 23

FIGURE 4: OPTIONAL QUALIFIED ASSOCIATION 24

FIGURE 5: ALTERNATE QUALIFIERS 24

FIGURE 6: TRANSITIVE CLOSURE INVOLVING GENERALIZATION AND ASSOCIATION [1] 26

FIGURE 7: SQL STATEMENT TO FIND TABLES WITH SSAN AS AN ATTRIBUTE 29

FIGURE 8: INITIAL OBJECT DIAGRAM BEFORE THE DATA ANALYSIS 34

FIGURE 9: BINARY ASSOCIATION 34

FIGURE 10: QUALIFIED ASSOCIATIONS 35

FIGURE 11: ER DIAGRAM (PERSON) 54

FIGURE 12: ER DIAGRAM (RESIDENT STUDENT 1) 55

FIGURE 13: ER DIAGRAM (RESIDENT STUDENT 2) 56

FIGURE 14: ER DIAGRAM (RESIDENT STUDENT 3) 57

FIGURE 15: PERSON'S OBJECT MODEL 58

FIGURE 16: PERSON OBJECT MODEL (CONT.) 59

FIGURE 17: RESIDENT STUDENT OBJECT MODEL 60

FIGURE 18: RESIDENT STUDENT OBJECT MODEL (CONT.) 61

FIGURE 19: GRADE HISTORY ASSOCIATION 61

FIGURE 20: GRADE CHANGE HISTORY ASSOCIATION 62

FIGURE 21: STUDENT COURSES ASSOCIATION 62

FIGURE 22: DROPPED COURSE ASSOCIATION 62

FIGURE 23: WAIVED COURSE ASSOCIATION 62

FIGURE 24: EDUCATION HISTORY ASSOCIATION 63

FIGURE 25: STARS APPLICATION LEVEL 0 DFD 82

FIGURE 26: PERFORM ACTION LEVEL 1 DFD 83

FIGURE 27: MENU OPTION LEVEL 2 DFD 84

FIGURE 28: HANDLE SELECTION LEVEL 2 DFD 85

FIGURE 29: PERFORM SELECTION LEVEL 3 DFD 86

FIGURE 30: PERFORM INSERTION LEVEL 4 DFD 87

FIGURE 31: PERFORM UPDATE LEVEL 4 DFD 88

FIGURE 32: PERFORM DELETION LEVEL 4 DFD 89

FIGURE 33: TABLE'S RELATIONSHIP IN STARS DATABASE 90

FIGURE 34: PERSON'S CLASS 91

FIGURE 35: MILITARY'S CLASS 92

FIGURE 36: MILITARY_STUDENT'S CLASS 93

FIGURE 37: MILITARY_RESIDENT_STUDENT'S CLASS 94

FIGURE 38: MILITARY_INTL_STUDENT'S CLASS 95

FIGURE 39: CIVILIAN'S CLASS 96

FIGURE 40: CIVILIAN_STUDENT'S CLASS 97

FIGURE 41: CIVILIAN_RESIDENT_STUDENT'S CLASS 98

FIGURE 42: CIVILIAN_INTL_STUDENT'S CLASS 99

FIGURE 43: ADDRESS' CLASS 100

AFIT/GCS/ENG/96J-01

Abstract

This research proposes and evaluates a methodology for reengineering a relational database to an object-oriented database. We applied this methodology to reengineering the Air Force Institute of Technology Student Information System (AFITSIS) as our test case. With this test case, we could verify the applicability of the proposed methodology, especially because AFITSIS comes from an old version of Oracle RDBMS. We had the opportunity to implement part of the object model using an object-oriented database, and we present some peculiarities encountered during this implementation. The most important result of this research is that it demonstrated that the proposed methodology can be used for reengineering an arbitrarily selected relational database to an object-oriented database. It appears that this approach can be applied to any relational database.

A METHODOLOGY FOR REENGINEERING RELATIONAL DATABASES TO AN OBJECT-ORIENTED DATABASE

1 *Introduction*

The software reengineering process has been used to solve many problems involving legacy systems. It has been helping companies to recover and to update documentation, design, and requirements of important systems. Most of the time thousands of lines of code are the only source of the business rules, and are the starting point in the process of reverse engineering. Software reengineering has been playing an important role and has been proven to be very effective in extending the lifetime of many applications.

All systems have a limited lifetime. Each implemented change erodes the structure which makes any following changes more expensive. As time goes on, the cost to implement a change will be too high, and the system will not be able to support its intended task. The reengineering process plays an important role by not allowing the system to reach this condition.

1.1 Background.

The goal of reengineering is to mechanically reuse past development efforts in order to reduce maintenance expense and improve software flexibility. Reengineering is applicable to diverse software such as programming code, databases, and inference logic [1].

There are many possible motives for the reverse engineering of databases [2]:

- **Migration between database paradigms.** One may want to migrate between database paradigms, for example from past hierarchical, network, and relational databases to modern relational and object-oriented databases;
- **Migration within a database paradigm.** A more mundane task would be to migrate between different implementations of a database paradigm, for example from one vendor's relational database to another relational database;
- **Documentation.** Reverse engineering can elucidate poorly documented existing software when the developers are no longer available for advice;
- **Tentative requirements.** Reverse engineering of existing software can yield tentative requirements for the new replacement system. Reverse engineering ensures that the functionality of the existing system is not overlooked or forgotten;
- **Assessment of software.** The quality of the database design is an indicator of the quality of the software as a whole. An understanding of the concepts supported by the underlying database schema allows one to better judge functionality claims;
- **Integration.** Reverse engineering facilitates integration of related legacy applications and purchased applications. A logical model of encompassed software is a prerequisite for integration;

- **Conversion of legacy data.** One must fully understand the logical correspondence between the old database and the new database before attempting to convert data.

1.2 Problem.

The main difficulty to reengineering relational databases is the lack of a robust process that can be applied in all cases. Most of the existing processes for database reverse engineering are inadequate; they assume too high a quality of input information [2].

1.3 Hypothesis.

The maintenance of a relational database application can be improved by:

1. Reverse engineering the system to develop an object-oriented model;
2. Redesigning the system using an Object-Oriented Methodology;
3. Changing the Database Management System to one that supports an object-oriented approach.

1.4 Research Objectives.

In order to solve the problem stated above and establish the validity of the above hypothesis, the following objectives were established:

1. Define an appropriate reverse-engineering methodology;
2. Determine an appropriate database application to be a test case;
3. Analyze and reverse engineer the test case using this methodology;
4. Redesign the test case using object-oriented methods;

5. Implement a portion of the new design in an Object-Oriented Database Management System prototype system;
6. Analyze the methodology based on this experience.

1.5 Test case.

With the intention of conducting directly useable research in the field of software reengineering, the director of the Communication Computer System of the Air Force Institute of Technology (AFIT/SC) was contacted. Discussions led to the discovery that his working group was facing a significant reengineering task which could be used as a basis test case for this thesis research.

In 1987 the Air Force Institute of Technology (AFIT) contracted the development of an automated system called Student Tracking and Registration System (STARS). This system is used for scheduling courses, registering students in courses, tracking academic histories of students, and generating related reports. The STARS application uses the Structured Query Language (SQL) to access an Oracle Relational Database Management System (RDBMS) Version 6. This system also uses the following tools: the SQL-Forms, SQL-ReportWriter, SQL-Menu, VMS, and Batch files [3]. From the time the system was designed until this thesis effort, requirements have been changing. Some of these changes were implemented, while others were not.

Even though this system is only eight years old, it is already considered old or a legacy system. This quick obsolescence was caused mainly by the following [4]:

1. Changes were made to incorporate some new requirements; however, documentation was not updated;

2. Past leaders who lacked software knowledge;

3. New technology;

4. Poor training;

5. Lack of focus on changing needs.

The Air Force Institute of Technology Student Information System (AFITSIS) was chosen as the test case in implementing a new method for reengineering relational databases to an Object-Oriented database.

AFITSIS is currently designed and implemented using relational technology and unfriendly user interface mechanisms. This old design and technology cause the maintenance to be difficult, because there are no maintainability features. This lack of maintainability demands a lot of time and effort every time new requirements are implemented on the system. Additionally, the system is inflexible and complex, requiring for each change up to five hundred forms and reports to be updated and checked for consistency.

1.6 Assumptions.

The following assumptions were made for the thesis research:

1. The decision to reengineer AFITSIS instead of starting the analysis and design of a new system is the best decision;

2. Access to AFITSIS and query information from the database are available;

3. Access to an Object-Oriented Database Management System (OODBMS) is available for use.

1.7 Sequence of Presentation.

The thesis is divided into six chapters. Chapter I, Introduction, has provided an overview of the work. Chapter II, Summary of Current Knowledge, discusses the background information which provides the foundation for this research. Chapter III, The Methodology, presents a proposed methodology for reengineering a relational database to an object-oriented database. Chapter IV, Application of the Methodology, presents the application of the proposed methodology using AFITSIS as a test case. Chapter V, Implementation Issues, discusses how the selected part of AFITSIS was implemented using an OODBMS. Lastly, Chapter VI, Analysis, Conclusions, and Recommendations, analyzes the results obtained from the application and implementation of the methodology, draws conclusions from this analysis, and makes recommendations for futures applications of this methodology.

2 *Summary of Current Knowledge*

2.1 Treatment and Organization.

This literature review provides the foundation to create a methodology for reengineering relational database applications to an object-oriented database. This chapter is divided into three sections: software reengineering, reengineering of relational databases, and object-oriented methodology. The software reengineering section gives an overview of the software reengineering process. The reengineering of relational databases section presents the basic steps when reverse engineering relational databases. The object-oriented methodology section describes the stages used by developers to analyze a problem, design a system, and implement the system into a usable product.

2.2 Software Reengineering.

Reengineering, also viewed as both renovation and reclamation, is the examination and alteration of a system to reconstitute it in a new form. Reengineering usually includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or re-structuring [5].

Reverse engineering is a process of examination and analysis of the subject to identify its components and create a higher level form of abstraction [5]. It can start at any stage of the life-cycle and it does not involve changes to the subject. Its sub-products include the design recovery and the redocumentation of the subject. Forward engineering can be easily understood as a process of moving from a high-level of

abstraction to low-level or physical details. It is the same as the traditional method of developing a new system. This term is used only to distinguish this process from reverse engineering. Figure 1 illustrates the basic ideas of software reengineering using, for simplicity, only three life-cycle stages of software.

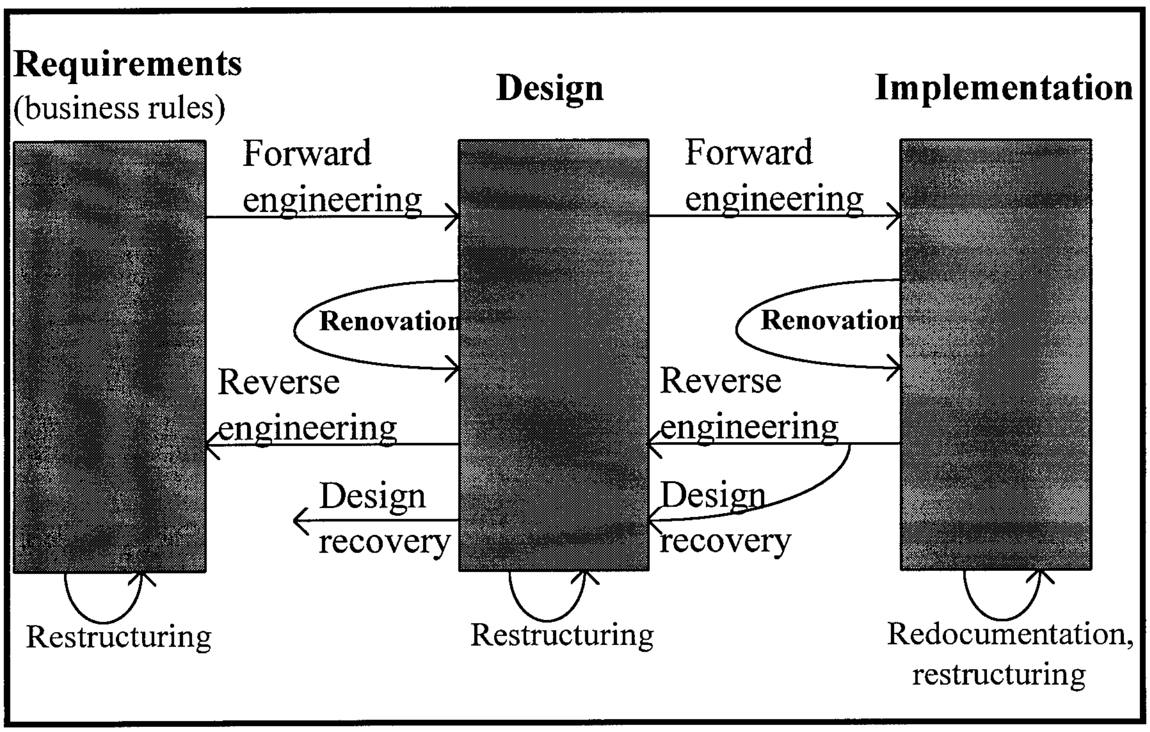


Figure 1: Relationship between terms [5]

The objectives of Software reengineering can be grouped into four main areas [6]:

1. ***Improve maintainability***. The maintenance efforts can be reduced by reengineering smaller modules with more explicit interfaces. However, it is not easy to measure progress toward this goal.
2. ***Migration***. This task usually deals with altering and converting program structure. This goal can be easily measured, since the system will perform the same operation in the new environment.
3. ***Achieve greater reliability***. This goal can be easily reached because the restructuring process usually causes most of the potential defects to appear. The other factor that contributes to better software reliability is the extensive testing required to prove the functional equivalence between the old and the new system. This goal can be readily measured by fault analysis.
4. ***Preparation for functional enhancement***. Once the programs are decomposed into smaller modules, it is easier to isolate them from one another. This makes it simpler to change or add new functions without affecting other modules.

2.3 Reengineering of Relational Databases.

The goal of reengineering is to mechanically reuse past development efforts in order to reduce maintenance expense and improve software flexibility. According to Hainaut [7] the most tractable approach for database applications is to first reverse

engineering the database and then deal with the programming code. Object-oriented models provide a natural language for facilitating the reengineering process. An object-oriented model can describe the existing software, the reverse-engineering semantic intent, and the forward-engineered new system.

In general, the mapping between object models and a database schema is many-to-many. Various optimizations and design decisions can be used to forward engineer an object model into a database schema. Similarly, when reverse engineering a database, alternate interpretations of the structure and data can yield different object models. Usually there is no obvious, single correct answer for reverse engineering. Multiple interpretations can all yield plausible results [2].

A good way to begin reverse engineering is by entering the existing schema into a CASE tool. Associations will often be found in a degraded form such as relational database foreign keys. Inheritance must be implemented in a degraded manner for current relational database managers. The schema may then be gradually transformed to a logical model as underlying relationships are inferred.

Jacobson [8] presents a good approach for reengineering old systems to an object-oriented architecture, but he does not give much information when dealing with relational databases. The same problem exists when considering other approaches for reengineering like those of Bennett [9] and Sneed [6]; they are not focusing on relational databases.

A good approach is suggested by Blaha [1], [2]. His papers present some typical implementation strategies that are used for forward engineering. He explains in

detail each step to be taken for reverse engineering of relational databases. The basic steps he suggests are:

Step 1. Prepare an initial object model.

- Represent each table as a tentative class. All columns of tables become attributes of classes.

Step 2. Determine candidate keys.

- Look for unique indexes. Automated scanning of data can yield potential candidate keys.

Step 3. Determine foreign-key groups.

- Try to resolve homonyms, attributes with the same name that refer to different things, and synonyms, attributes with different names that refer to the same thing.
- Matching attribute names, data types, and/or domains may suggest foreign keys.
- During this step do not attempt to determine specific reference-referent attribute pairs – but merely groups of attributes within which foreign keys may be found.

Step 4. Refine tentative classes.

- Agglomerate horizontally partitioned classes into a single class. (horizontally partitioned classes must also have the same semantic intent.)
- Detect functions and constraints that are represented as tables.

Step 5. Discover generalizations.

- Analyze large foreign-key groups, particularly those with 5, 10, or more cross-related attributes.
- Look for patterns of many replicated attributes.
- Look for patterns of data where a class has mutually exclusive subsets of attributes.
- When discovering generalizations do not forget there may be a forest of generalizations with multiple superclass roots and intermediate levels.

Step 6. Discover associations.

- Convert a tentative class to an association when a candidate key is a concatenation of two or more foreign keys.
- Introduce a qualified association when a candidate key combines a foreign key with non-foreign key attributes.
- The remaining associations are buried and manifest as foreign keys.
- Note minimum multiplicity for associations. Optional multiplicity is the permissive case; a lower limit of one (or another number) is more restrictive.
- Note maximum multiplicity for associations. Many multiplicity is the permissive case; an upper limit of one (or another number) is more restrictive.

- Apply semantic understanding and restate some associations as aggregations. Aggregation is the "a-part-of" relationship.

Step 7. Perform transformation.

- Convert a class to a link class as needed.
- Lightweight one-to-one associations should be more simply represented as an attribute.
- Nonatomic n-ary associations should be decomposed into their constituent associations of lesser order.
- Consider shifting associations via transitive closure.
- Double-buried associations should be merged into a single association.
- You may need to insert an intermediate class in a generalization hierarchy to recognize common semantics, attributes, and associations.
- Transitive closure also arises through the combination of generalization and association. Where possible, eliminate an imprecise association to a superclass in favor of a more restrictive association to a subclass.
- Similarly, eliminate associations to subclasses by recognizing patterns of commonality.

2.4 Object-Oriented Methodology.

One of the primary reasons for adopting object technology is the promise of faster development and reduced maintenance costs. In traditional systems, ongoing maintenance costs amount to more than 80% of the overall cost of the system [10]. Object-oriented systems promise to reduce maintenance costs through reusable objects that can dramatically reduce maintenance. In many cases, developers only need to identify an object class that functions like the object that they desire to create, and specify the differences between the object and their new object. This type of code reusability can dramatically reduce development and maintenance costs.

Object-oriented methodology allows developers to analyze problems and divide them into entities residing in specific states and exhibiting certain dynamic behaviors. The entities become objects in the system. The designer defines the relationships between the objects to determine how the system functions as a whole. The four specific stages of object-oriented methodology are [11:4-6]:

1. ***Analysis.*** During the analysis stage, the developer defines the system requirements. Objects are identified and their relationships to other objects are recorded. There are no implementation decisions in this stage. Three models are defined in this stage: an object relationship model, a dynamic model, and a functional model;

2. ***System Design.*** In this stage the system's architecture is determined. The application is broken into subsystems. Control mechanisms are defined for

each subsystem. The focus is on what needs to be done, and not how it is to be done;

3. ***Object Design.*** During this phase, the object relationship model, dynamic model, and functional model are evaluated to determine what operations must be implemented for each object. Structures for representing the relationships between objects are defined.

4. ***Implementation.*** The final stage involves transforming the design into an executable system. This is dependent on whether the software language selected supports object-oriented programming.

2.5 Conclusion

This literature review has provided an overview of the basic concepts of software reengineering, the reengineering of relational databases, and object-oriented methodology. All three of these areas are required for the successful analysis and implementation of the new methodology.

3 *Methodology*

3.1 Introduction

This chapter presents the methodology for reengineering a relational database to an object-oriented database. It shows the methodology step by step explaining each step in detail, including some discussion of typical implementation techniques that one can find during the process of reverse engineering.

This methodology is based on Blaha [1] [2] with some changes. His papers were selected because they are focused specifically on reverse engineering of a relational database to an object-oriented database and they are the only ones that give detailed information on this subject.

Some changes were introduced on his approach just to facilitate the transition from relational to an object-oriented view. The most important changes are:

1. Construct an entity-relationship model instead of going directly from the tables to an object model;
2. Besides the object model, prepare a functional model to facilitate the implementation of the system.

3.2 The Methodology

This methodology is presented in a linear fashion for ease of understanding, but, except for the first and last step, the others steps are weakly ordered since during the process of reverse engineering there is much iteration and backtracking. The steps are as follows:

Step 1. **Prepare an entity-relationship (ER) model.**

This step can be easily accomplished by using an automated tool. Otherwise proceed as listed below:

- Represent each table as an entity.
- Determine candidate keys. Look for unique indexes, but some candidate keys may not be enforced by unique indexes. Automated scanning of data can yield potential candidate keys.
- Determine primary keys. Ordinarily every table should have a primary key. But exceptions can be encountered as follow:

  1. Tables with temporary data or tables which the performance overhead can not be tolerated.
  2. Missing primary key without cause. Some applications enforce primary keys with custom code and do not rely upon the database manager.
  3. Null primary key attributes. Some relational database managers require that one define a unique index to enforce a primary key. Indexed attributes are permitted to be null, unless "not null" is specified for each of the attributes. This violates the definition of primary key; attributes in a primary key may not be null.

4. Extraneous primary key attributes. By definition a primary key must also be minimal; no attribute can be discarded from the primary key without destroying uniqueness. The reverse engineer must regard all primary key declarations with suspicion, and look for attributes that do not seem semantically justified.

Even when tables do have a primary key, different realizations may still be chosen. Figure 2 shows relational tables for three different approaches to identify the primary key. All three schemas can be reverse engineered to the same logical model.

- **Artificial identity**. Each object table (shown in Figure 2) has an object identifier as primary key. Association tables (not shown in Figure 2) have a primary key consisting of the identifiers of the related objects.

- **Value-based identity**. The primary key of each object consists of some combination of application attributes. Some primary keys may become lengthy, as attributes are incorporated from foreign key of related tables.

- **Hybrid identity**. One may use artificial identity and value-based identity in the same schema. In the third segment of Figure 2 *Bank* has artificial identity and *Account* has identity derived from a reference to a bank combined with an account number.

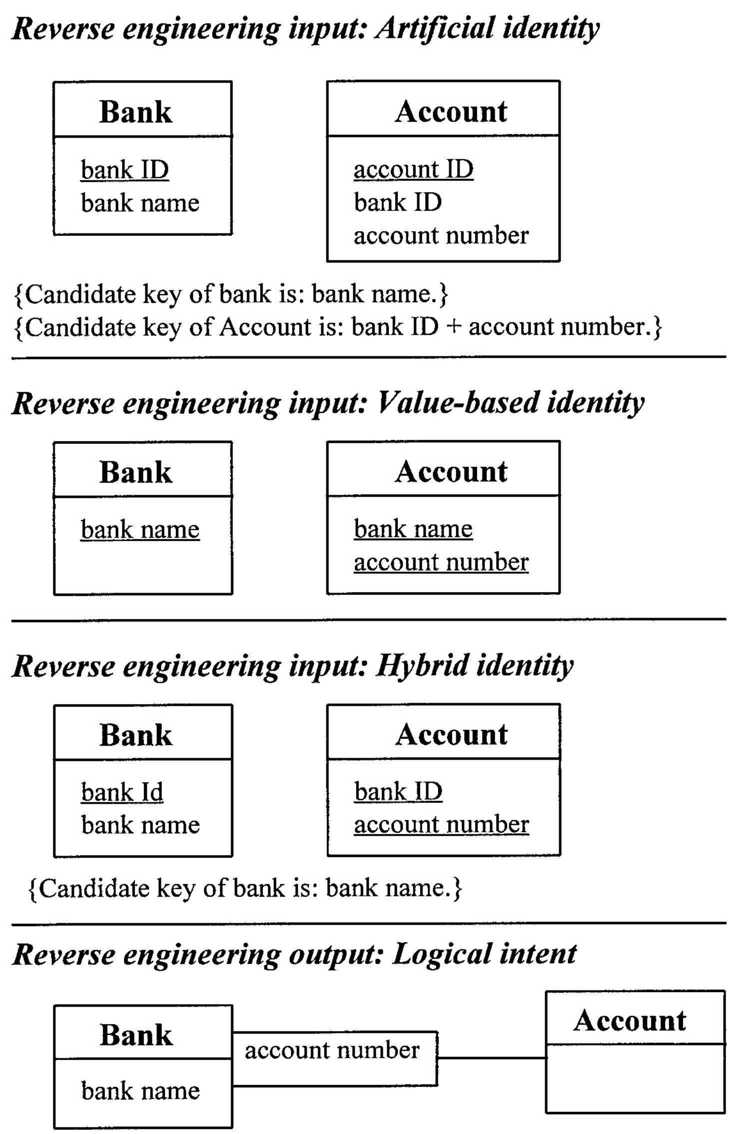


Figure 2: Various approaches to identify the primary key [2]

- Determine foreign-keys. Most of the modern RDBs have a foreign-key clause as part of the schema. If you do not have this do the following:
    - Try to resolve homonyms, attributes with the same name that refer to different things, and synonyms, attributes with different names that refer to the same thing.

- Matching attribute names, data types, and/or domains suggest foreign keys.

- Generate the relationships by checking every possible foreign key against every candidate key.

- Finish the ER model by querying the data and determining the multiplicity of each relationship.

Step 2. **Prepare an initial object model.**

Based on the ER diagram, represent each entity as a tentative class and each relationship as a tentative association. All columns of the related tables become attributes of classes.

Step 3. **Refine tentative classes.**

Agglomerate horizontally partitioned classes into a single class. Horizontally partitioned classes have the same schema. Distributed databases often use horizontal partitioning to disperse records. (Horizontally partitioned classes must also have the same semantic intent. Identical schema is a good indicator of same semantic intent.)

Detect functions and constraints that are represented as tables and take these classes out of the tentative object model. Look for classes that do not participate in any foreign key.

Step 4. **Discover generalizations.**

Analyze large foreign-key groups, particularly those with 5, 10, or more cross-related attributes. Look for a primary key that is entirely composed of a foreign key of another table. Derived identity is symptomatic of an implementation of generalization with distinct superclass and subclass tables or propagation of identity via one-to-one association. Data analysis can increase confidence in the discovery of generalization by revealing subsets of records.

Look for patterns of many replicated attributes. A generalization may have been implemented by pushing superclass attributes down to each subclass.

Look for patterns of data where a class has mutually exclusive subsets of attributes. This may indicate an implementation of generalization where subclass attributes were pushed up to the superclass.

When discovering generalizations one must not forget there may be a forest of generalizations with multiple superclass roots and intermediate levels. Data analysis can help distinguish multiple, disjoint, and overlapping inheritance. (Keep in mind that data analysis only yields hypotheses, and semantic understanding is required to reach firm conclusions.)

Step 5. **Discover associations.**

Convert a tentative class to an association when a candidate key is a concatenation of two or more foreign keys. Where possible, try to restate ternary and

n-ary associations (confluence of primary keys from three or more classes) as binary associations[2].

Introduce a qualified association when a candidate key combines a foreign key with non-foreign key attributes. This will find some, but not all, qualifiers.

The remaining associations are buried and manifest as foreign keys.

Note minimum multiplicity for associations. Optional multiplicity (nulls allowed) is the permissive case as for a given record you may store an actual value or store a null; a lower limit of one (or another number) is more restrictive.

Note maximum multiplicity for associations. Many multiplicity is the permissive case as a collection can store a single value or many values; an upper limit of one (or another number) is more restrictive.

Apply semantic understanding and restate some associations as aggregations. (Aggregation is the "a-part-of" relationship.)

When discovering associations be aware to the following kind of implementations that one may encounter [2]:

- **Double-buried associations**. This is when an association was buried in both participating classes as shown in Figure 3. This construct complicates reverse engineering, since these double-buried associations look like two separate associations. Data analysis can detect redundancy between the dual pointers, but semantic understanding is required to resolve this situation.

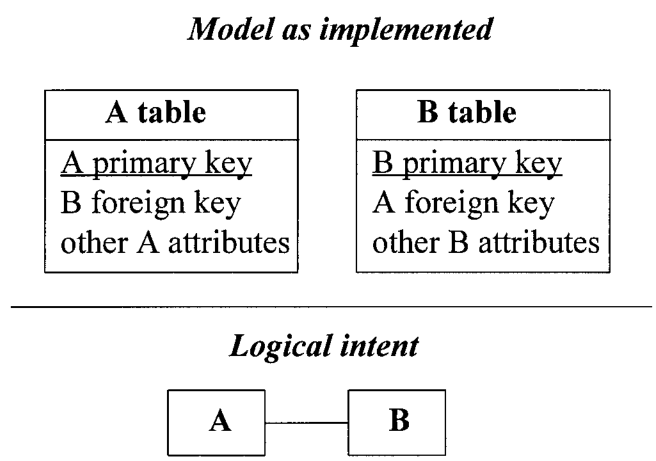


Figure 3: Double buried association

- **Optional qualified association.** Figure 4 shows an optional qualified association. A *cluster* contains many *Tables*. A *Table* may belong to at most one *Cluster*. The combination of a *Cluster* and a *table#* yields a specific *Table*. This association was implemented by burying *cluster_id* as a foreign key in *Table*. Because of the optional membership in a cluster, the foreign key can be null, and the combination of *cluster_id* and *table#* is not a candidate key of *Table*. Therefore it is difficult to detect this qualified association.

- **Alternate qualifier**. In Figure 5 *Column* derives its identity from a *Table* plus a qualifier, either *column name* or *column number*.

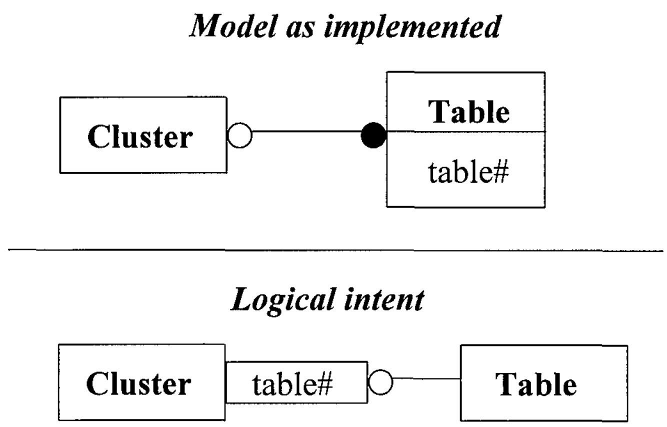


Figure 4: Optional qualified association

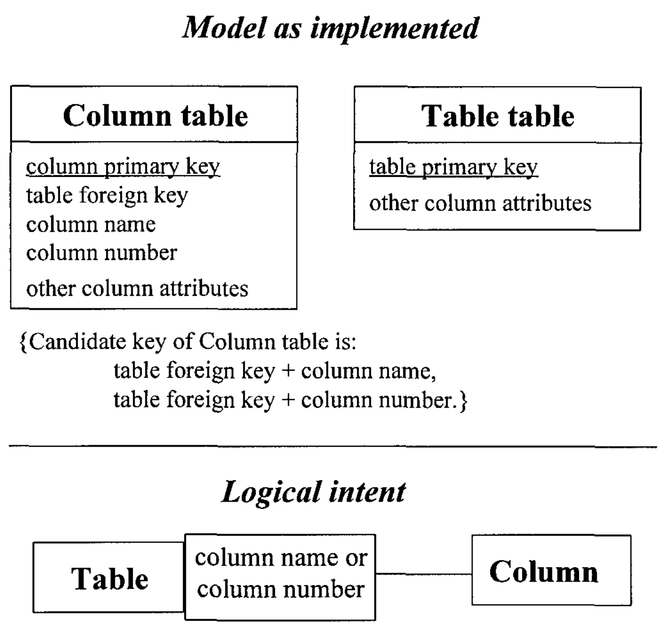


Figure 5: Alternate qualifiers

***Step 6*. Perform transformation.**

Various optimizations may have been employed in preparing the original RDB schema to improve time and/or space performance. Some transformations are listed here [1].

- Convert a class to a link class as needed. A link class is an association whose links can participate in associations with other classes. An association has derived, rather than intrinsic, identity.

- Lightweight one-to-one associations (they have no attributes) should be more simply represented as an attribute. For example, it is unnecessary to represent **city** as a class, when **city-name** is the only attribute of interest.

- Nonatomic n-ary associations should be decomposed into their constituent associations of lesser order. Binary associations are most common and easier to understand. We may find ternary association, but never an association of higher order.

- Consider shifting associations via transitive closure. For example associations from **A** to **B** and **B** to **C** could possibly be restated as associations from **A** to **B** and **A** to **C**. In general, multiplicity constrains derivation of association, but the vague multiplicity limits often obtained through reverse engineering allow more latitude.

- Double-buried associations should be merged into a single association. For example, an association between **A** and **B** may have been buried in both the **A** and **B** classes.

- You may need to insert an intermediate class in a generalization hierarchy to recognize common semantics, attributes, and associations.

- Transitive closure also arises through the combination of generalization and association. Where possible, eliminate an imprecise association to a

superclass in favor of a more restrictive association to a subclass. For example, in Figure 6 if our semantic knowledge is that **X** only associates with **B** and never with **C**, then we can eliminate the association between **X** and **A** and the association between **X** and **C**.

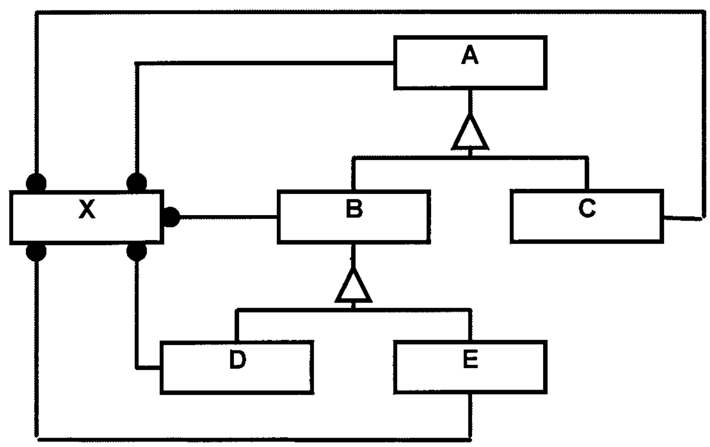


Figure 6: Transitive closure involving generalization and association [1]

- Similarly, eliminate associations to subclasses by recognizing patterns of commonality. In Figure 6, if all instances of **B** partition across classes **D** and **E**, we can eliminate the association between **X** and **D** and the association between **X** and **E**.

Step 7. **Prepare a functional model.**

The functional model describes computations within a system, and specifies the results of this computation without specifying how or when they are computed. Database system often have a trivial function model, since their purpose is to store and organize data, not to transform it [11:123].

One can prepare the functional model only using the user manual, the forms, and, if necessary, interviewing the users.

3.3 Summary

This chapter has presented the methodology for reengineering a relational database to an object-oriented database. This methodology is heavily based on Blaha papers [1] and [2], except for the first step, that was introduced to facilitate the transition from relational to an object-oriented view, and the last step, that was introduced to give more information about the functionality of the system. It showed each step to be followed with some discussion of typical implementation techniques. The next chapter presents the application of this methodology using AFITSIS as a test case.

4 *Application of Methodology*

4.1 Introduction

This chapter presents the application of the proposed methodology using AFITSIS as a test case. It is divided into three sections: the first section shows how the ER model was obtained. The next section presents the transformations that were made to the ER model to obtain the object model. The last section shows how the functional model was drawn.

Following direction of the sponsor (AFIT/SC), this analysis is restricted to those tables and forms that have some relationship to the *Person* table. This restriction does not invalidate the work, since about 66 of 294 tables from the entire AFITSIS are considered.

4.2 ER Model

To accomplish the first step of the methodology, which is to draw the ER model, we used ERwin (an ER diagram editor developed by Logic Works [12]). Since AFITSIS was developed for Oracle version 5, which does not support foreign-key clauses, and migrated to Oracle version 6 without changes, ERwin was able to capture only the tables and its attributes (all 294 tables from AFITSIS). If you are reverse engineering a RDB that supports foreign-key clauses, ERwin can recover not only the tables and their attributes, but also foreign-keys, the relationships between tables, and can draw the entire ER model.

We started our work identifying the *Person's* primary key (SSAN). Next we selected all tables that have this primary key as an attribute by querying the Oracle data dictionary (Figure 7). This query resulted in 66 tables (Appendix A).

```
SELECT table_name
FROM accessible_columns
WHERE column_name
like "%SSAN%";
```

Figure 7: SQL statement to find tables with SSAN as an attribute

The next step was to determine the candidate keys for each of the selected tables. We looked for unique indexes in the data dictionary, and for each one that we found we scanned the data to confirm the correctness. For the other tables for which we could not find a unique index, we had to scan the data.

During the process of scanning the data to look for the primary key, we were able to find many tables that have no data, tables that have not been used for many years, and tables that were used as a temporary files. After we confirmed that they were not being used by any form, we eliminated these tables from our diagram.

We looked for foreign key groups by matching attribute names and types. We did not have any complication in this step, especially because the names are very suggestive and we did not find any homonyms nor synonyms.

Next, we were able to generate the relationships between the entities by checking every possible foreign key against every candidate key, and linking the

related entities using ERwin. In our model we did not consider as a relationship the link of a table with a validation table, via its foreign key. For example: *Address* table has as foreign keys the attributes *Address_type_code, Street_type_code, Address_room_type_code,* and *country_code*; which are the primary key of the following validation tables (table look up): Address_type_valid, Street_type_code_valid, Address_room_type_code_valid, and Country_valid, respectively.

To finish the ER model (see Appendix B) we queried the data to determine the multiplicity of each relationship, doing the following:

- **One-to-one association**. To determine a one-to-one association we verified if for each row in one table of the relationship there was only one entry into the other table.

- **One-to-one-or-more association**. For this type of association we verified whether for each row in one table we found at least one or more entries into the other table.

- **One-to-zero-or-one association**. In this case we verified whether for each row in one table we could find zero or exactly one entry into the other table.

- **One-to-zero-one-or-more association**. Now we verified whether for each row in one table we found zero or more entries into the other table.

4.3 Object Model

To draw the object model we started preparing an initial object diagram (step 2) based on the ER diagram, where we represented each entity as a tentative class and each relationship as a tentative association. We transformed all columns of the related entity into the attributes of the class.

We started refining the object model (step 3) by looking for horizontally partitioned classes (classes with the same schema) and representing them as a single class. This is what we found:

- The classes *Term_entry* and *Term_entry_history* had the same schema and the same semantic intent. We merged them into a single class;
- The classes *Selected_student*, *Selected_student_91*, and *Selected_student_new_91*, had the same schema but, after checking the data, we determined that the classes *Selected_student_91* and *Selected_student_new_91* were used as temporary files. We retained only the class *Selected_student* and eliminated the others.

Then we looked for tables that could have been representing functions and/or constraints, but we did not find any.

We started the process of discovering generalization (step 4) by looking for large foreign-key groups. Although we found a couple of tables in this case we

realized that these tables were not involved in a generalization but in a binary or ternary association.

Generalization was found when we started looking for any class that had its primary key entirely composed of a foreign key of another class. We took these classes apart and analyzed their relationship.

To do a good analysis of this kind of relationship we had to improve our semantic knowledge of the system. We did that by making some queries and analyzing its results, by looking up the forms, and by interviewing the Database Administrator (DBA). After that, we were able to take these classes and select those involved in a generalization from those involved in an association. For example: the tables *Spouse_info, Emergency_data, AFIT_user_name, Recall_roster, Dependent_information, and Graduation_name* all have their primary key entirely composed of *Person's* primary key, but they have no inheritance relationship with this table.

Semantic knowledge was especially important to incorporate some abstract classes. For example, in the object diagram in Figure 15 (Appendix C), the abstract classes *Civilian* and *Military* were introduced after we discovered that *Faculty*, *Student*, and *Administrative* people could be either military or civilian, but only military people have a relationship with *Rank_history* and *Recall_roster*. So, we decided to introduce these two abstract classes to increase code reuse and to organize features common to these subclasses.

Another generalization chain was encountered when we analyzed *Student*, *CI_student*, *Resident_student,* and *INTL_student* tables. We found out that every instance of *CI_student* and *Resident_student* was in the *Student* table, and that every instance of *INTL_student* was in the *Resident_student* table.

We introduced the class *Part-time_student* to represent another kind of student, after we discovered that this class was implemented as an attribute of *Resident_student* class called *program_code.* One of the valid values is 'PTE', meaning 'part-time student'.

Data analysis was very important to increase confidence in the discovery of generalization. Figure 8 shows our initial object diagram where we made some assumptions based on our understanding of the system. But, after we analyzed the data in the *Eligible*, *Selected_students*, and *Student* tables, we discovered that, contrary to our assumption, not all instances of *Student* could be found in the *Selected_student* table, and that not all instances of *Selected_student* could be found in the *Eligible* table. This data analysis led us to change our object diagram to the one shown in Appendix C.

We continued our work of drawing the object model by discovering other associations (step 5). We started this step by looking for candidate keys composed of two or more foreign keys, and we converted it into an association. Figure 9 shows one binary association that we found. All the other associations can be seen in Appendix C.

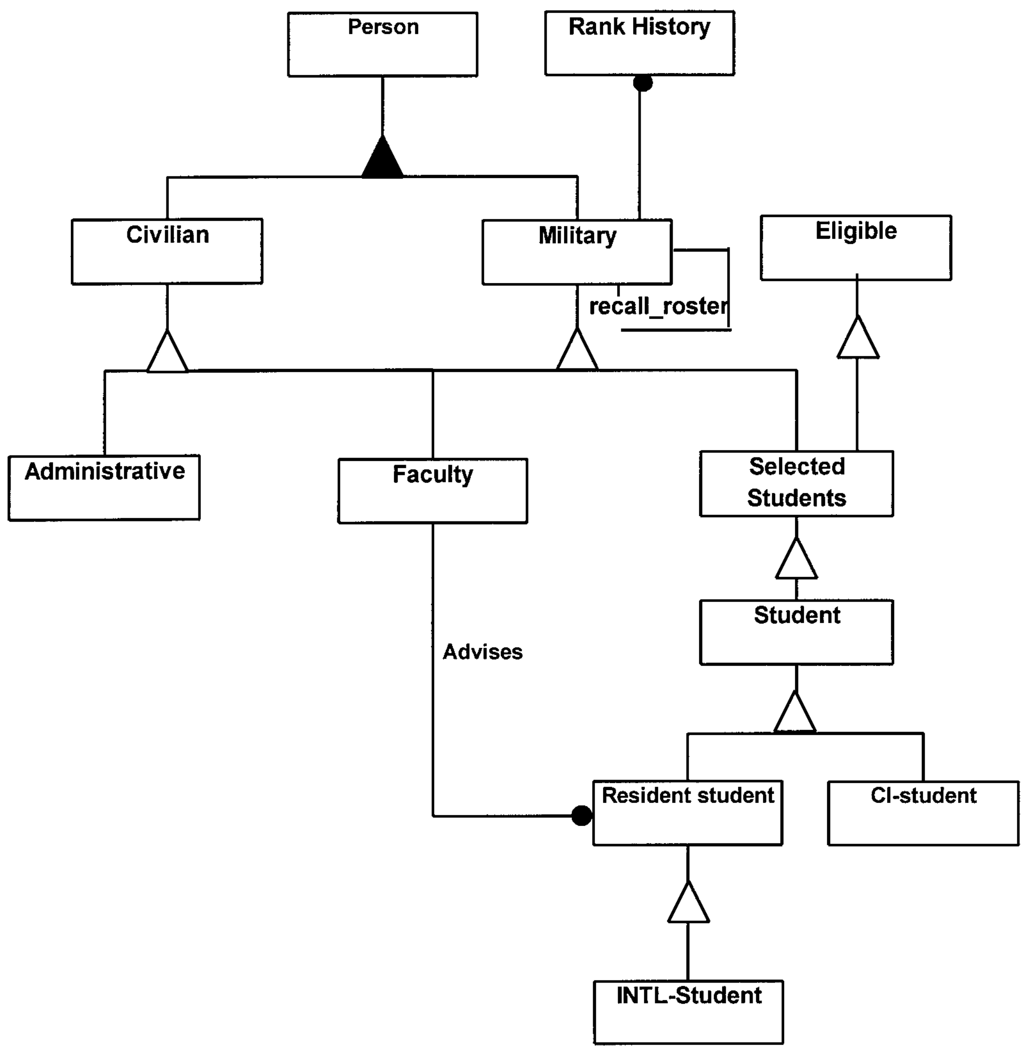


Figure 8: Initial object diagram before the data analysis

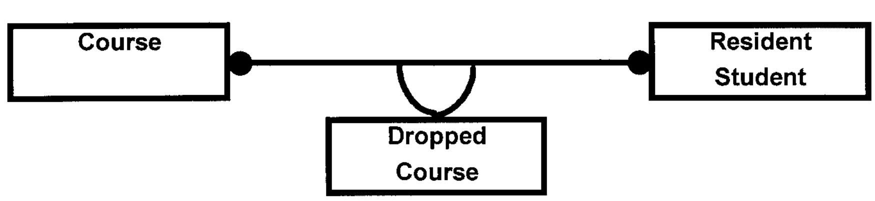


Figure 9: Binary association

We introduced a qualified association when we found a candidate key that combined a foreign key with a non-foreign key attribute. Figure 10 shows some of the qualified associations that we introduced to reduce the effective multiplicity of the association (from many to one), to improve semantic accuracy, and to increase the visibility of navigation paths.

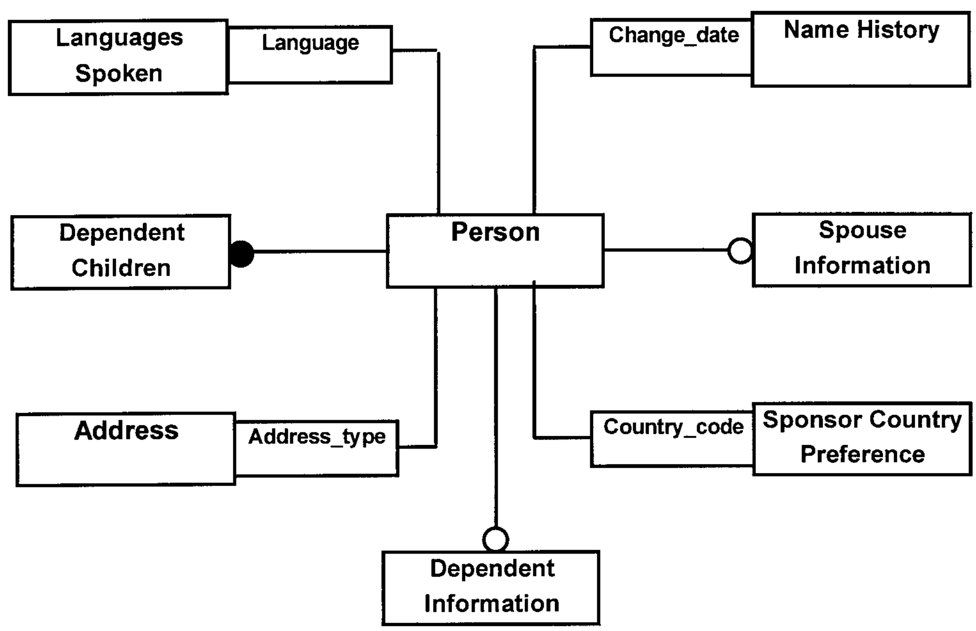


Figure 10: Qualified associations

In our research we were not able to find the following cases of association:

- Association that could be representing an aggregation;
- Double-buried associations;
- Optional qualified association;

- Alternate qualifier.

We started our final step (step 6-perform transformation) to get the object model by transforming each lightweight one-to-one association into an attribute. We found a one-to-one association of *Person* with *AFIT_user_name* and *Person* with *Emergency_data* and we transformed these associations into attributes of *Person* (Person Structure Definition, Appendix C). We did the same with a one-to-one association of *Resident_student* with *Graduation_name* (see Resident Student Structure Definition, Appendix C).

We did not find any class that could be better represented as a link class. This does not mean that none of our classes is a link class; the only restriction is that our view is limited, since our research is concerned with only part of AFITSIS. We did not have to do any work to decompose n-ary associations into their constituent associations of lesser order, especially because we had only binary associations. Our complete object model, including the object structure definition, can be seen in Appendix C.

4.4 Functional Model

In order to get the functional model done we initially made use of the STARS User's Guide [3] to select the boundaries of what we were going to implement, and then limited our work in doing the functional model to this specific part. We used SQLForms to extract the name of the tables that each form can read or update, what

actions the form is doing, and all the other information needed to draw the functional model. The complete functional model is shown in Appendix D.

4.5 Summary

This chapter has presented the application of the proposed methodology using AFITSIS as a test case. It showed how the ER model was obtained, the transformations that were made on the ER model to get the object model, and how the functional model was drawn. The next chapter discusses the implementation of part of AFITSIS using an OODBMS.

5 *Implementation Issues*

5.1 Introduction

This chapter presents the implementation issues concerning the development of part of AFITSIS using an OODBMS. It is divided into three sections: the first section discusses how we analyzed several OODBMS and why we chose Microsoft Visual Foxpro (Foxpro) version 3 to be used in this implementation. The next section shows how part of the object model was implemented. The last section discusses some limitations encountered when using Foxpro as an OODBMS.

5.2 Analysis and Choice of the OODBMS

During the process of choosing one OODBMS to implement part of AFITSIS we took the following considerations:

- Based on the interview [4], AFIT/SC wanted to migrate AFITSIS to an OODBMS. However, before making any decision, they will wait for Oracle Company to release version 8, expected to be an extension of the RDBMS with some object capabilities;

- Since Oracle version 8 is not expected to reach the market until the end of 1996 or beginning of 1997, and with the intention of doing a useful implementation, we looked for an available RDBMS that would have some similarities with the expected Oracle version 8. These similarities that we were concerned about are: the product should be able to use all the power

and flexibility of RDBMS, like Data Definition Language (DDL) and Data Manipulation Language (DML), share the basic relational tables, and incorporate some concept of "object," and have the ability to store procedures as well as data in the database.

With these considerations in mind we started analyzing some available OODBMS. The first two OODBMS that we analyzed were ITASCA and Objectstore. Even though we concluded that each is a very good OODBMS, we decided not to use either of them because they are heavily based on some language like C or C++, and they have no compatibility and similarity with any RDBMS.

The next product that we analyzed was Foxpro. Once we had some experience in using an old version of Foxpro and knowing that it is a RDBMS, we concentrated our analysis to see if the new object-oriented features would be compatible with what we wanted. After we read the Foxpro Developer's Guide [13] and used it for two weeks, we were convinced that this product could give us a good means of comparison and insight of what we could expect when we have the Oracle version 8 available.

5.3 Implementation of the Object Model

We started the implementation of our object model by creating a new project and inserting a new database that we called *Stars*. In Foxpro the terms *database* and *table* are not synonymous. The term *database* refers to a relational database that stores information about one or more *tables* or views [13]. The *database* is where we can

create stored procedures (that can be used as field- and record-level rules) and persistent table relationships (to enforce referential integrity).

After we created the *Stars* database we created the definition of the tables that we were going to use, their primary key and indexes, and we added these tables to the database. Then we linked the tables to set up the relationships (Figure 33, Appendix E), so that we do not need a code program to check the referential integrity every time an application tries to modify the database. The database manager system takes care of it whenever the database is opened and used.

The next step was to create the forms, one for each table. After that, we were ready to start creating the definitions of the classes. To implement the Person's Object Model (Figure 15, Appendix C) we did the following:

- We created the P*erson's* class based on the P*erson's* form (Figure 34, Appendix E);
- We created the *Military's* class based on the *Person's* class and adding the *Military's* form (Figure 35, Appendix E);
- We created the *Military_student's* class based on the *Military's* class and adding the *Student's* form (Figure 36, Appendix E);
- We created the *Military_faculty's* class based on the *Military's* class and adding the *Faculty's* form;

- We created the *Military_resident_student's* class based on the *Military_student's* class and adding the *Resident_student's* form (Figure 37, Appendix E);
- We created the *Military_INTL_student's* class based on the *Military_Resident_student's* class and adding the *INTL_student's* form (Figure 38, Appendix E);
- We created the *Civilian's* class based on the *Person's* class and adding the *Civilian's* form (Figure 39, Appendix E);
- We created the *Civilian_student's* class based on the *Civilian's* class and adding the *Student's* form (Figure 40, Appendix E);
- We created the *Civilian_faculty's* class based on the *Civilian's* class and adding the *Faculty's* form;
- We created the *Civilian_resident_student's* class based on the *Civilian_student's* class and adding the *Resident_student's* form (Figure 41, Appendix E);
- We created the *Civilian_INTL_student's* class based on the *Civilian_resident_student's* class and adding the *INTL_student's* form (Figure 42, Appendix E);

The reason we created the classes this way was to give more flexibility and to make the classes easier to maintain. For example: if we need to make some change in

person's class we do not have to modify all the other classes that use it. Because of the inheritance feature, the changes that we make in the parent class reflect over the subclasses automatically; and if we need to overload some parent method (function, procedure, trigger, or event) so that it takes a different action when running the subclass, it can be easily achieved by just creating a method in the subclass with the same name as the parent class. This way the subclass method will have precedence over the parent class method.

We implemented binary associations by first creating a view with the related tables and then creating a form based on this view. For example: the association between *Person* and *Address* (Figure 16, Appendix C) was implemented by creating a view with *Person* and *Address* tables, and then creating a form using this view, the *Person* class, and the *Address* form (Figure 43, Appendix E).

We implemented a binary association with link attributes by creating a new table with these link attributes and the primary key of the two associated tables as foreign keys. We created a form based on this new table, and a class based on this form. For example: the Dropped Courses Association (Figure 22, Appendix C) was implemented by creating a *Dropped_courses* table having the primary key of the *Course* and *Resident_student* tables as foreign keys, plus the link attributes. Then we created a form base on the *Dropped_courses* table and a class *Dropped_courses* based on this form.

5.4 Limitations of Foxpro Encountered During Implementation

During the implementation of our object model we faced two major problems when using Foxpro as an OODBMS. The first one is that in Foxpro we can't define one class based on a table by only adding its methods. Instead, we have to define the table, define one form based on this table, and then define the class based on this form and add the methods. Actually, this peculiarity does not cause much of a problem (when you need to modify some attribute, you have to change the table structure and its related form), but it is a little different from what we learned in theory [11].

Another problem encountered is that Foxpro did not appear to support multiple inheritance. To implement the *Person's* Object Model (Figure 15, Appendix C), instead of defining only one class for *Student*, *Faculty*, and *Administrative*, we had to define the classes *Military_student*, *Military_faculty* *Military_administrative*, *Civilian_student*, *Civilian_faculty*, *Civilian_administrative*, and so on. This restriction may cause some problems if the subclass has other relationships. For example: the relationship *Advises* between *Faculty* and *Resident_student* (Figure 15, Appendix C), has to be implemented by defining one relationship *Advises* from *Civilian_faculty* to *Civilian_resident_student* and *Military_resident_student*, and the same relationship *Advises* from *Military_faculty* to *Civilian_resident_student* and *Military_resident_student*.

5.5 Summary

This chapter has presented the implementation issues of part of AFITSIS. Some OODBMS were analyzed and Foxpro was chosen because of its similarity to what we are expecting for Oracle version 8. We showed the implementation techniques that we used to implement inheritance and some associations. During the implementation we found two limitations in using Foxpro as an OODBMS. One is that we can't define a class directly from a table and the other is that it does not support multiple inheritance. With this implementation done, we had the last piece of information necessary to make an analysis and conclusion of our research. That is presented in the next chapter.

6 *Analysis, Conclusions, and Recommendations*

6.1 Analysis of The Results

In Chapter III we presented a methodology for reengineering a relational database to an object-oriented database. To validate this methodology we applied it to reengineering AFITSIS as a test case. As we presented in Chapter IV, this methodology is easy to use in practice. We did not have any difficulty when following its steps.

Our methodology has the purpose of reengineering a relational database, independent of the kind of the RDBMS and its version. With our test case, we had the opportunity to verify this applicability, especially because AFITSIS comes from an old version of Oracle RDBMS. This way we could apply most of the steps of what we proposed in the methodology. For example: to accomplish the first step of the methodology, which is to draw the ER model, we used ERwin. Since AFITSIS was developed for Oracle version 5, which does not support foreign-key clauses, ERwin was able to capture only the tables and their attributes. Since ERwin was not able to draw the entire ER model and facilitate this job, we really had to apply the methodology and follow its steps to recover the foreign-keys and the relationships between tables.

When applying our methodology to draw the object model we found out that semantic logic can play an important role, especially to discover generalization and to

incorporate some abstract classes. Another important factor that we found that increased our confidence in the discovery of generalization was data analysis. After we analyzed the data we could change our first object diagram to the one shown in Appendix C. Even though in our test case we were not able to apply our methodology to exemplify the discovery of all kinds of associations, we were able to find some of them.

The most important result of this analysis is that it demonstrated that the proposed methodology can be easily used for reengineering any relational database to an object-oriented database, filling the lack of a robust process that can be applied in all cases.

6.2 Conclusion

The life span of an information system consists of specification, design and maintenance. The maintenance phase dominates in time and often with respect to resources as well. During this phase the system is subjected to a number of changes and additions. The gap between the older technology in the system and the new technology that becomes available increases successively. Changes in the activities of an organization also mean that systems grow old.

Gradually the system approaches a limit where it no longer is cost-efficient or even technically feasible to continue the maintenance. But the cost of enforcing the required changes is usually very high [8]. A possible way out of this dilemma is to

define well delimited system parts that are candidates for modernization. This is where reengineering can help.

We have described a practical method for reengineering. The method is based on object-oriented modeling. We have described how the work can be divided into a number of steps so that the method can be performed in a systematic manner.

We have used AFITSIS as a test case and have shown that with this method we can model an existing system in a simple manner and with limited effort. The new model is object-oriented and can serve as a basis for a future development plan.

We have implemented part of AFITSIS using Foxpro, one OODBMS that we have chosen because of its similarities with the expected Version 8 of Oracle. From this experience we were able to see that our object model can be easily mapped to be implemented using another OODBMS.

The six research objectives, as stated in Chapter I, were:

1. Define an appropriate reverse-engineering methodology;
2. Determine an appropriate database application to be a test case;
3. Analyze and reverse engineer the test case using this methodology;
4. Redesign the test case using object-oriented methods;
5. Implement a portion of the new design in an Object-Oriented Database Management System prototype system;
6. Analyze the methodology based on this experience.

The research was successful in all the original objectives. We presented a practical methodology that can be applied for reengineering any relational database system. We chose AFITSIS as a test case, we applied our methodology for reengineering it, we obtained an object and functional model from this work, and we implemented this model using an OODBMS. Finally, one of the most important lessons that we have learned when working with reengineering is that in general, the mapping between object models and schemes are many-to-many. Various optimizations and design decisions can be used to transform an object model into a database schema. Similarly, when reverse-engineering a database, alternate interpretations of the structure and data can yield different object models. Usually, there is no obvious, single correct answer for reverse engineering. Multiple interpretations can yield plausible results [1].

6.3 Recommendations

For those who intend to use the object model obtained from our test case, we recommend that you revise this model making another data analysis. This is because we had restricted access to AFITSIS tables, since they record confidential information. Doing this you can have more confidence on the model, and may find some important information that we were not able to uncover.

Even though Foxpro was demonstrated to be a good OODBMS to be used in our test case, I recommend further analysis concerning its security of the data. This is because security is an important aspect to be considered for adopting an OODBMS to

implement AFITSIS, and Foxpro does not appear to have any mechanism to restrict the access to the databases (for example: password with level of access.)

Appendix A: List of Tables with SSAN

(Pk): Primary key; (Fk): Foreign key.

Address: SSAN (Pk) (Fk), Address_Type_Code (Pk), Address_Line_1, Address_Line_2, Address_Line_3, City, State_Code, Zipcode, Zipcode_Extension, Country_Code, Area_Code, Phone_Number, Address_Effective_Date, DSN_Prefix, Login_Name, Firm_Name_Office_Symbol, Additional_Address_Information, Street_Address, Street_Type_Code, Address_Room_Type_Code, Address_Room_Type_Number, Revision_Name, Revision_Date, Country, Login_Date, Phone_Number_Ext.

Address_Data_Final: SSAN (Pk) (Fk), Address_Type_Code (Pk),Firm_Name_Office_Symbol, Additional_Address_Information, Street_Address, City, State_Code, Zipcode, Zipcode_Extension, Street_Type_Code, Address_Room_Type_Code, Address_Room_Type_Number, Area_Code, Phone_Number, Address_Effective_Date, DSN_Prefix, Login_Name, Revision_Name, Revision_Date, Country.

AFITnet_User_Name: SSAN (Pk) (Fk), Login_Name, Input_Date, User_Name, User_UID, Host_Accnt_Created

Ci_Student: SSAN (Pk) (Fk), Major00career_Pointer_Code, Academic_Status_Code, Accounting_Status_Code, Book_Payment_Authorize_Code, Ci_Student_Comment, Civilian_Institution_Code, Corps_Code, Course_Of_Study, Current00ASC_Code, Current00ewi_Option_Code, Email_Address, Grad_Date, Gre_Status_Code, Hpsp_Medical_Code, Ida_Date, Kit_Sent_Date, MAJCOM_Abbrev, Major00ASC_Code, Motorcycle_Status_Code, Motorcycle_Train_Date, Office_Code , Overseas_Indicator, Program_Entry_Date, Quota_Program_Code, Report_No_Earlier_Than_Date, Report_No_Later_Than_Date, Residence00state_Code, Residency_Status_Code, Selected00ASC_Code, Selected00career_Pointer_Code, Selected00ewi_Option_Code, Selected_Date, Selected_Quota_Year, Thesis_Diss_Required_Code, Thesis_Program_Complete_Date, Type_Degree_Code, Input_Date, Login_Name, Ewi00occupation_Series_Code, Scholarship_Type_Code, DLI_Language_Code, Af_Acad_Sponsor_Dept_Code, No_Cost_Indicator, Esp_Code, Book_Qtrs_Paid, DLI_Entry_Date, School_In_Civ_Ins_Code, Remarks, Advance_Flag.

Class_Leader: SSAN (Pk) (Fk), Leader_Code (Pk), Program_Graduation00Term_Code, Class_Code, Program_Year_Prefix.

Degree_Awards: AFIT_Degree_Code (Pk) , SSAN (Pk) (Fk), Career_Pointer_Code, Grad_Status_Code, Pse_Code, Grade_Rank_Abbrev, Name_Prefix, Name_Suffix, First_Name , Last_Name , Middle_Initial, Birth_Date, Sex_Code, Race_Code, Marital_Status_Code, Religion_Code, Blue_Chip_Indicator, Aka_Fname, Aka_Lname, Prior_AFIT_Months, Tafms_Date, Ethnic_Group_Code, Aero_Rating_Code, Manning_Code, Deros_Date, Separation_Date, Commission_Code, Grade_Rank_Date, Citizenship00country_Code, Department_Code, Duty_Title, Duty_Phone, Duty_Area_Code, Badge_Number, Academic_Action_Code, Overdue_Indicator, Classification_Code, Part_Record_Indicator, Admin_Hold_Indicator, Major00ASC_Code, Academic_Specialty, Major00ed_Level_Code, Ed_Level, Program_Code, Program, Program_Graduation00term_Code, Class_Code, Program_Year_Prefix, Selected_Type_Code, Selected_Type, AFIT_Degree, Graduation00term_Code, Graduation00quarter_Code, Graduation_Year_Prefix, Graduation_Date, Departure_Date, Box_Number, Card_Number, Encoded_Card_Number, Library_Number, Locker_Number, Admit_Date, Student_Sponsor_SSAN, Entry00term_Code, Entry00quarter_Code, Entry_Year_Prefix, Admission_Type_Code, Admission_Action_Code, Gaining00AFSC_Code, Faculty_Advisor_SSAN, Registration00department_Code, Program_Effective00term_Code, Effective00quarter_Code, Effective_Year_Prefix, Leader_Code, Program_Section_Number,

Gain00MAJCOM_Abbrev, Gain00duty_Station, Losing00MAJCOM_Abbrev, Double_Degree_Indicator, Branch_Service_Code.

Dependent_Children: SSAN (Pk) (Fk), Child_Last_Name (Pk), Child_First_Name (Pk), Dependent_Child_Birth_Date, Dependent_At_AFIT_Indicator, Child00sex_Code.

Dependent_Information: SSAN (Pk) (Fk), Number_Children, Sngle_Dep_Chldrn_Indicator, Deps_At_AFIT_Indicator.

Drop_Table: Course_Prefix_Code (Pk) (Fk), Course_Number (Pk) (Fk), Course_Section (Pk) (Fk), SSAN (Pk) (Fk), Course_Dropped_Date, Drop_Reason.

Edplan_Desc: Career_Pointer_Code (Pk), SSAN (Pk) (Fk), Description, Description_Line_Number.

Education_History: MPC_School_Code (Pk) (Fk), Ed_Level_Code (Pk), SSAN (Pk) (Fk), Type_Degree_Code, ASC_Code, Quality_Points, Total_Credit_Hours, Method_Of_Obtainment_Code, Academic_Ed_Status_Code, Input_Date, Operators_Initials, Login_Name, Last_Year_Attended, ABET_Accredited_Indicator, Ed_History_Remarks, Work_ID_Processed_Code, Trnscrpt00career_Pointer_Code, Duty_Location_Code, Degree_Cum_Gpa, Degree_Title.

Eligibility: SSAN (Pk) (Fk), Eligibility_Evaluation_date (Pk), Pre_AFIT00Ed_Level_Code (Pk), Counse;or_Initials, Elig_Overall_GPA, Elig_Math_GPA, Elig,Major_GPA, Evaluation_Status_Code, List_Number.

Emergency_Data: SSAN (Pk) (Fk), Emergency_Contact_Fname, Emergency_Contact_Lname, Emergency_Relation, Address_Line_1, Address_Line_2, Address_Line_3, City, State_Code, Zipcode, Zipcode_Extension, Country_Code, Area_Code, Phone_Number, Country, Firm_Name_Office_Symbol, Additional_Address_Information, Street_Address, Address_Room_Type_Code, Address_Room_Type_Number, Street_Type_Code, Revision_Name, Revision_Date, Login_Name, Login_Date.

EN_Program_Leader: SSAN (Pk) (Fk),EN00Leader_Code, EN_Student00Program_Code, Program_Graduation_Term_Code, Class_Code, Program_Year_Prefix.

Evaluation_By_School: SSAN (Pk) (Fk), Admitted_Indicator, Evaluation_Result_Remark, Evaluation_Forwarded_Date, Forwarded_To00Department_Code, Evaluation_Returned_Date.

Faculty: SSAN (Pk) (Fk), AFIT_School_Code, Academic_Instruction_Indicator, Appointment_Type_Remark, Faculty_Type_Code.

Faculty_History: SSAN (Pk) (Fk), Academic_Rank_Code (Pk), Academic_Rank_Date, Academic_Step.

Fitness_Performance: SSAN (Pk) (Fk), Fitness_Category_Code(Pk), Elapsed_Time, Trial_Time, Input_Date, Login_Name, Distance.

Grade_Change_History: Course_Prefix_Code (Pk) (Fk), Course_Number (Pk) (Fk), Course_Section (Pk) (Fk), SSAN (Pk) (Fk), Term_Code, Grade_Effective_Date, Prior00grade_Code.

Grade_History: Course_Prefix_Code (Pk) (Fk), Course_Number (Pk) (Fk), Course_Section (Pk) (Fk), SSAN (Pk) (Fk), Term_Code, Approval_Code, Approval_Date, Career_Pointer_Code, Credit_Hours, Earned_Hours_Indicator, Gpa_Indicator, Grade_Code, Grade_Effective_Date, Login_Name, Grade_Type_Code, Input_Date, Prior00grade_Code.

Graduation_Attendees: SSAN (Pk) (Fk), Graduation00term_Code (Pk), Graduation00quarter_Code, Graduation_Year_Prefix.

Graduation_Date_History: SSAN (Pk) (Fk), Graduation00term_Code, Effective00term_Code, Grad_Status_Code, Departure_Date, Graduation00quarter_Code, Graduation_Year_Prefix, Effective00quarter_Code, Effective_Year_Prefix, Login_Name, Input_Date.

Graduation_Name: SSAN (Pk) (Fk), Graduation_Name.

Intl_Student: SSAN (Pk) (Fk), WSCN, ITO, Case_Number, DLI_Req_Indicator, DLI_Indicator, Evaluation_Request_Date, Requested00program_Code, Eval_Forward_Date, Forward_To00department_Code, Eval_Returned_Date, Admission_Status_Code, Eval_Remarks, Country_Notified_Date, AFSAT_Notified_Date, AFSAT_Quota_Indicator, First_Sponsor_SSAN, Second_Sponsor_SSAN, Source_Of_Funds_Code, AFSAT_Country_Code.

IP_Attendee: SSAN (Pk) (Fk), IP_Activity_Code (Pk), Activity_Date (Pk).

Languages_Spoken: SSAN (Pk) (Fk), Language_Code (Pk).

LS_Part_Time: SSAN (Pk) (Fk), PT_LS_Student00Program_Code (Pk).

LS_Section_Leader: SSAN (Pk) (Fk), Section_Number (Pk), LS00Leader_Code (Pk).

Majors: Career_Pointer_Code (Pk), Major (Pk), SSAN (Pk) (Fk), Login_Name, Input_Date.

Name_History: SSAN (Pk) (Fk), Name_Change_Date (Pk), First_Name, Last_Name, Middle_Initial, Name_Suffix, Name_Prefix, Login_Name, Marital_Status_Code.

New_AFSC: SSAN (Pk) (Fk), AFSC_Code (Pk), Prefix.

OER_Data: SSAN (Pk) (Fk), Last_OER_Date, OER_Due_Date.

PCE_Grade: SSAN (Pk) (Fk), PCE_Couse_Prefix (Pk), PCE_Couse_Number (Pk), PCE_Couse_Letter (Pk), PCE_Couse_Year, PCE00Grade_Code.

PCE_Std: SSAN (Pk) (Fk), PCE_Stay_Begin_Date (Pk), PCE_Stay_End_Date, PCE00Billeting_Code, MAJCOM_Code.

Person: SSAN (Pk), Grade_Rank_Abbrev, Name_Prefix, Name_Suffix, First_Name , Last_Name , Middle_Initial, Birth_Date, Sex_Code, Race_Code, Marital_Status_Code, Religion_Code, Blue_Chip_Indicator, Aka_Fname, Aka_Lname, Prior_AFIT_Months, Tafms_Date, Ethnic_Group_Code, Aero_Rating_Code, Manning_Code, Deros_Date, Separation_Date, Commission_Code, Grade_Rank_Date, Citizenship00country_Code, Department_Code , Duty_Title, Duty_Phone, Duty_Area_Code, Badge_Number, Branch_Service_Code, Login_Name, Input_Date, Duty_Phone_Ext.

Personnel: SSAN (Pk) (Fk), Personnel00Department_Code, Personnel_Hire_Date, Personnel_Duty_Title, Phone_Number.

PHD: SSAN (Pk) (Fk), PHD_Major_Remark, PHD_Minor_Remark.

Planes_Flown: SSAN (Pk) (Fk), Plane_Name.

Program_History: Program_Code (Pk), Program_Graduation00term_Code (Pk), Program_Effective00term_Code (Pk), SSAN (Pk) (Fk), Input_Date, Faculty_Advisor_SSAN, Class_Code, Program_Year_Prefix, Ed_Level_Code, ASC_Code, Login_Name, Effective00quarter_Code, Effective_Year_Prefix, Career_Pointer_Code, AFIT_Degree_Code.

Program_STD_Sections: SSAN (Pk) (Fk), Section_Number (Pk).

Rank_History: SSAN (Pk) (Fk), Grade_Rank_Abbrev (Pk), Grade_Rank_Date, Login_Name, Input_Date, Manning_Code, Branch_Service_Code.

Recall_Roster: SSAN (Pk) (Fk), Home_Phone_Number, Next_In_Chain_SSAN.

Registration_Verification: SSAN (Pk) (Fk), Term_Code, Quarter_Code, Year_Prefix, Registration_Notice.

Resident_Student: SSAN (Pk) (Fk), Academic_Action_Code, Overdue_Indicator, Classification_Code, Part_Record_Indicator, Admin_Hold_Indicator, Major00ASC_Code, Major00ed_Level_Code, Program_Code , Program_Graduation00term_Code, Class_Code , Program_Year_Prefix, Selected_Type_Code, AFIT_Degree_Code, Graduation00term_Code , Graduation00quarter_Code , Graduation_Year_Prefix, Grad_Status_Code, Departure_Date, Box_Number, Card_Number, Encoded_Card_Number, Library_Number, Locker_Number, Admit_Date, Student_Sponsor_SSAN, Entry00term_Code, Entry00quarter_Code, Entry_Year_Prefix, Admission_Type_Code, Admission_Action_Code, Career_Pointer_Code, Gaining00AFSC_Code, Faculty_Advisor_SSAN, Registration00department_Code , Program_Effective00term_Code, Effective00quarter_Code, Effective_Year_Prefix, Leader_Code, Program_Section_Number, Gain00MAJCOM_Abbrev, Gain00duty_Station, Losing00MAJCOM_Abbrev, Pse_Code.

Section_Leaders: Leader_Code (Pk), SSAN (Pk) (Fk), Program_Code, Class_Code, Program_Section_Number.

Selected_Comments: SSAN (Pk) (Fk), Selected_Comment.

Selected_Projection: SSAN (Pk) (Fk), Gain00MAJCOM_Code, Gain00MAJCOM_Abbrev, Gain_MAJCOM_Supervisor, Gain_MAJCOM_Supervisor_Phone, Gain_MAJCOM_DSN_Prefix, Gain_MAJCOM00Department_Code, Position_Number_Projected.

Selected_Student: SSAN (Pk) (Fk), Selected00ASC_Code, Selected00ed_Level_Code, Selected_Quota_Year, Selected00ewi_Option_Code, Pca_Indicator, Quota_Program_Code, List_Number, Report_No_Earlier_Than_Date, Report_No_Later_Than_Date, Projected_Start_Date, Assign_Avail_Date, Assign_Reason_Code, MAJCOM_Abbrev, Projected_Entry_Class, Selected_Date, Input_Date, Login_Name, Selected_Type_Code, Selected00AFSC_Code, Reselection_Code, MPC_School_Code, Pse_Code, Assign00department_Code.

Selected_Student_Archive: Selected_Quota_Year (Pk), SSAN (Pk) (Fk), Selected00ASC_Code, Selected00ed_Level_Code, Selected00ewi_Option_Code, Pca_Indicator, Quota_Program_Code, List_Number, Report_No_Earlier_Than_Date, Report_No_Later_Than_Date, Projected_Start_Date, Assign_Avail_Date, Assign_Reason_Code, MAJCOM_Abbrev, Projected_Entry_Class, Selected_Date, Input_Date, Login_Name, Selected_Type_Code, Selected00AFSC_Code, Gain00MAJCOM_Code, Gain00MAJCOM_Abbrev, Gain_MAJCOM_Supervisor, Gain_MAJCOM_DSN_Prefix, Gain_MAJCOM_Supervisor_Phone, Gain_MAJCOM00department_Code, Position_Number_Projected, Reselection_Code, Pse_Code, MPC_School_Code, Assign00department_Code.

Sponsors_Country_Prefere: SSAN (Pk) (Fk),Ce, Preferred00country_Code (Pk).

Spouse_Info: SSAN (Pk) (Fk), Spouse_Birth_Date , Spouse_Fname , Spouse_Lname, Spouse_At_AFIT_Indicator, Spouse_Nickname, Spouse_In_Military_Indicator, Spouse_Occupation, Spouse_Remarks.

Student: SSAN (Pk) (Fk), Last_Name, First_Name, Department_Code, Grade_Rank_Abbrev, Program_Code, Class_Code, Grad_Term, Date_Entered, Revision_Date.

Student_Address: SSAN (Pk) (Fk), Address_Type (Pk), Address_Type_Desc, Address_Line_1, Address_Line_2, City, State, ZIP, ZIP_Ext.

Student_Courses: Course_Prefix_Code (Pk) (Fk), Course_Number (Pk) (Fk), Course_Section (Pk) (Fk), SSAN (Pk) (Fk), Term_Code, Hours, Grade, Calculated_Field.

Student_Duty_History: Duty_Sequence_Number (Pk), SSAN (Pk) (Fk), Duty_Title, Duty00AFSC_Code, Duty_Organization, Duty_Station, Duty_Assigned_Date, Login_Name.

Student_Sequences: Program_Sequence_Code (Pk), SSAN (Pk) (Fk).

Term_Entry: SSAN (Pk) (Fk), Entry00term_Code, Entry00quarter_Code, Entry_Year_Prefix, Admission_Type_Code, Admission_Action_Code.

Term_Entry_History: SSAN (Pk) (Fk), Entry00term_Code, Entry00quarter_Code, Entry_Year_Prefix, Admission_Type_Code, Admission_Action_Code.

Test_Scores: Test_Type_Code (Pk), SSAN (Pk) (Fk), Test_Taken_Date, Test_Score, Login_Name, Input_Date.

TDY_Attendees: SSAN (Pk) (Fk), Left_For_TDY_Date (Pk), Returned_From_TDY_Datee, TDY_Destination_Code, TDY_Purpose.

Thesis_Diss_Book_Allowance: SSAN (Pk) (Fk), Allowance_Code, ASC_Code, ED_Level_Code.

Transcript_SSANS_105643: SSAN (Pk) (Fk), Last_Name, First_Name, Middle_Initial, Name_Suffix, Grade_Rank_Abbrev, Program_Code, Selected_Type_Code, Table_Indicator.

Transcript_Sent: SSAN (Pk) (Fk), Transcript_Sent_Date (Pk), Num_Transcript_Sent.

Transfer_Transcript: SSAN (Pk) (Fk), Course_Prefix_Code (Pk), Course_Number (Pk), Transfer_Course_Prefix (Pk), Transfer_Course_Number (Pk), Course_Section, AFIT00Credit_Hours, Earned_Hours_Indicator, GPA_Indicator, Transfer_Credit_Hours, Transfer00Grade_Code, Transfer_Start_Date, Transfer_End_Date, MPC_School_Code, Soche_Indicator, Trnsfr00Term_Code, Trnsfr00Quarter_Code, Trnsfr_Year_Prefix, Trnsfr00Career_Pointer_Code, Transfer_Course_Title, Transcrip_Course_Title.

Wait_List: SSAN (Pk) (Fk), Course_Prefix_Code (Pk), Course_Number (Pk), Course_Section (Pk).

Waived_Course: Waived00course_Prefix_Code (Pk), Waived00course_Number (Fk), Course_Prefix_Code (Pk) (Fk), Course_Number (Pk) (Fk), Course_Section (Pk) (Fk), SSAN (Pk) (Fk), Waived00grade_Code, Waived_Date.

Appendix B: The Entity Relationship Diagram

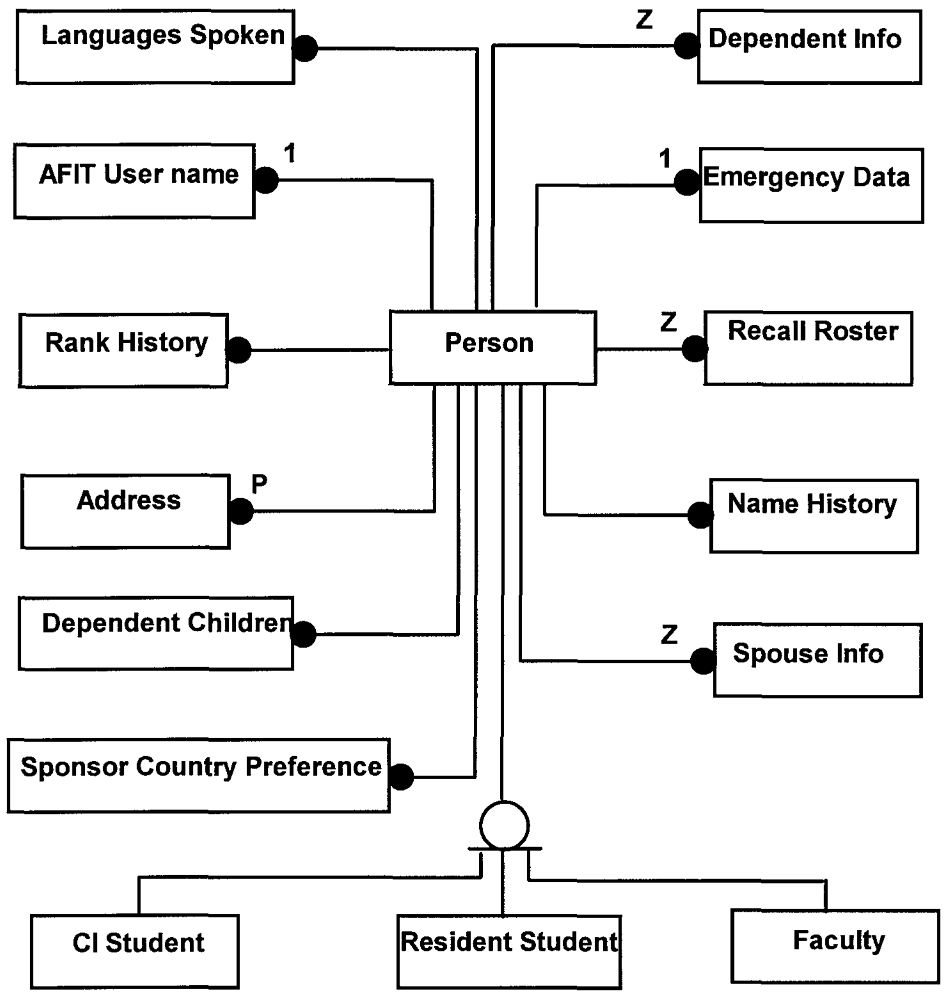


Figure 11: ER diagram (Person)

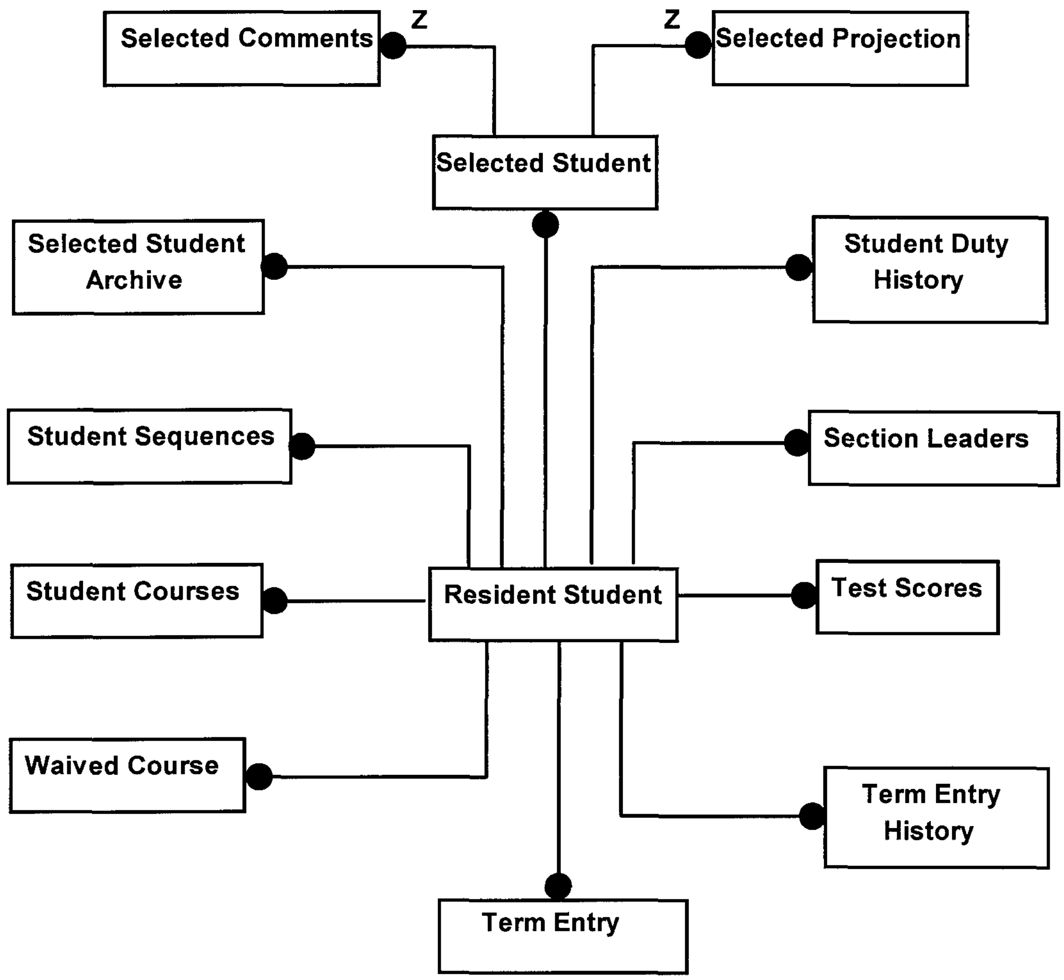


Figure 12: ER diagram (Resident Student 1)

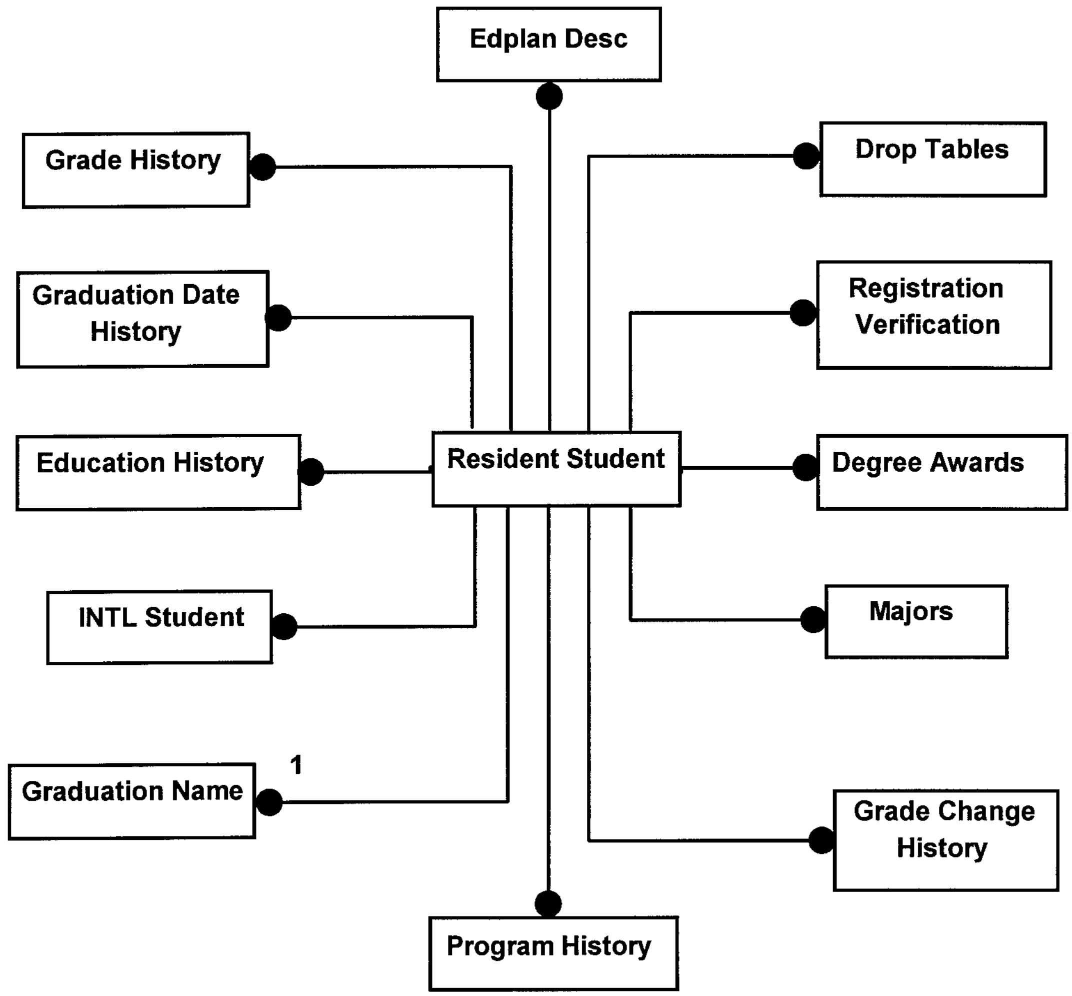


Figure 13: ER diagram (Resident Student 2)

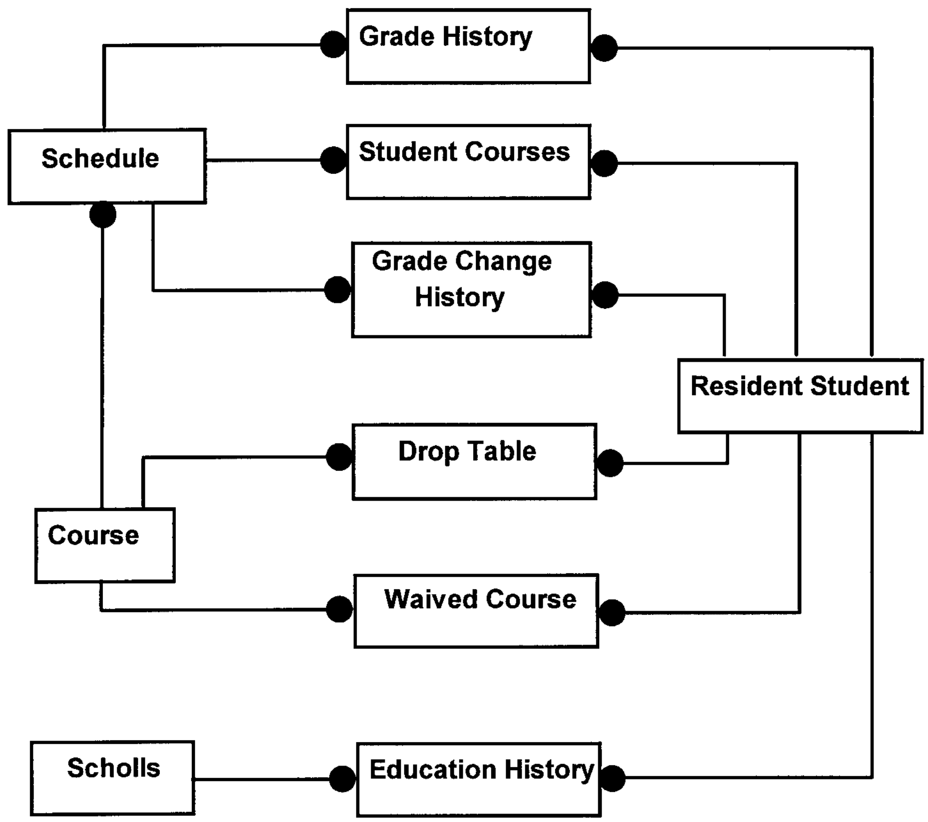


Figure 14: ER diagram (Resident Student 3)

Appendix C: The Object Model

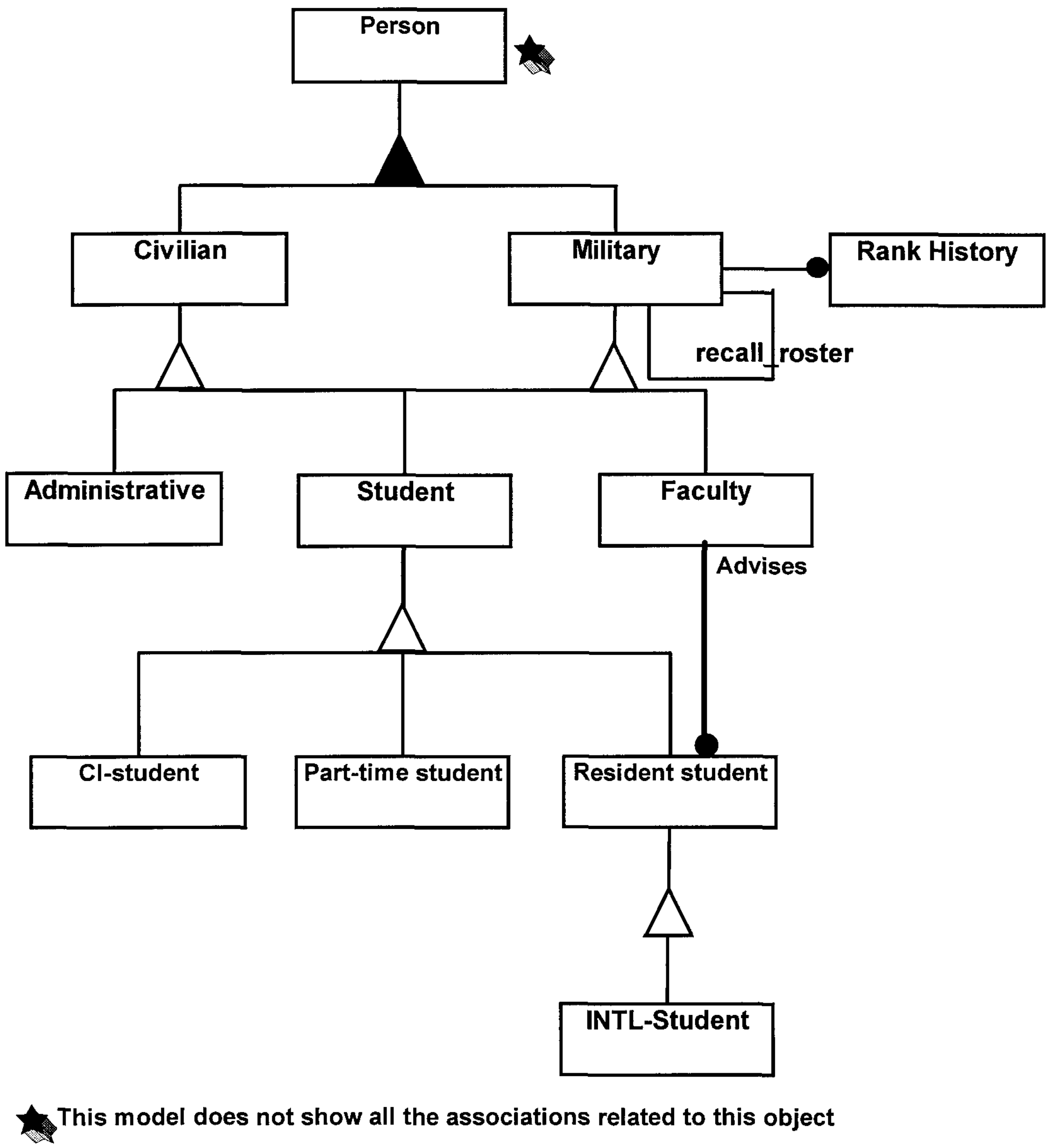


Figure 15: Person's Object Model

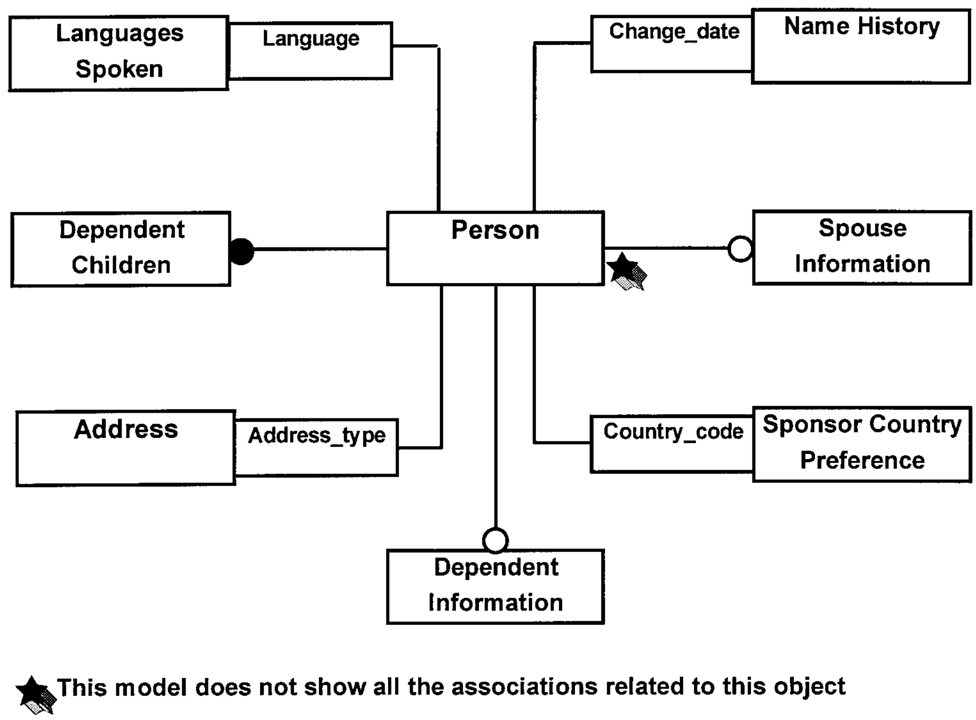


Figure 16: Person Object Model (cont.)

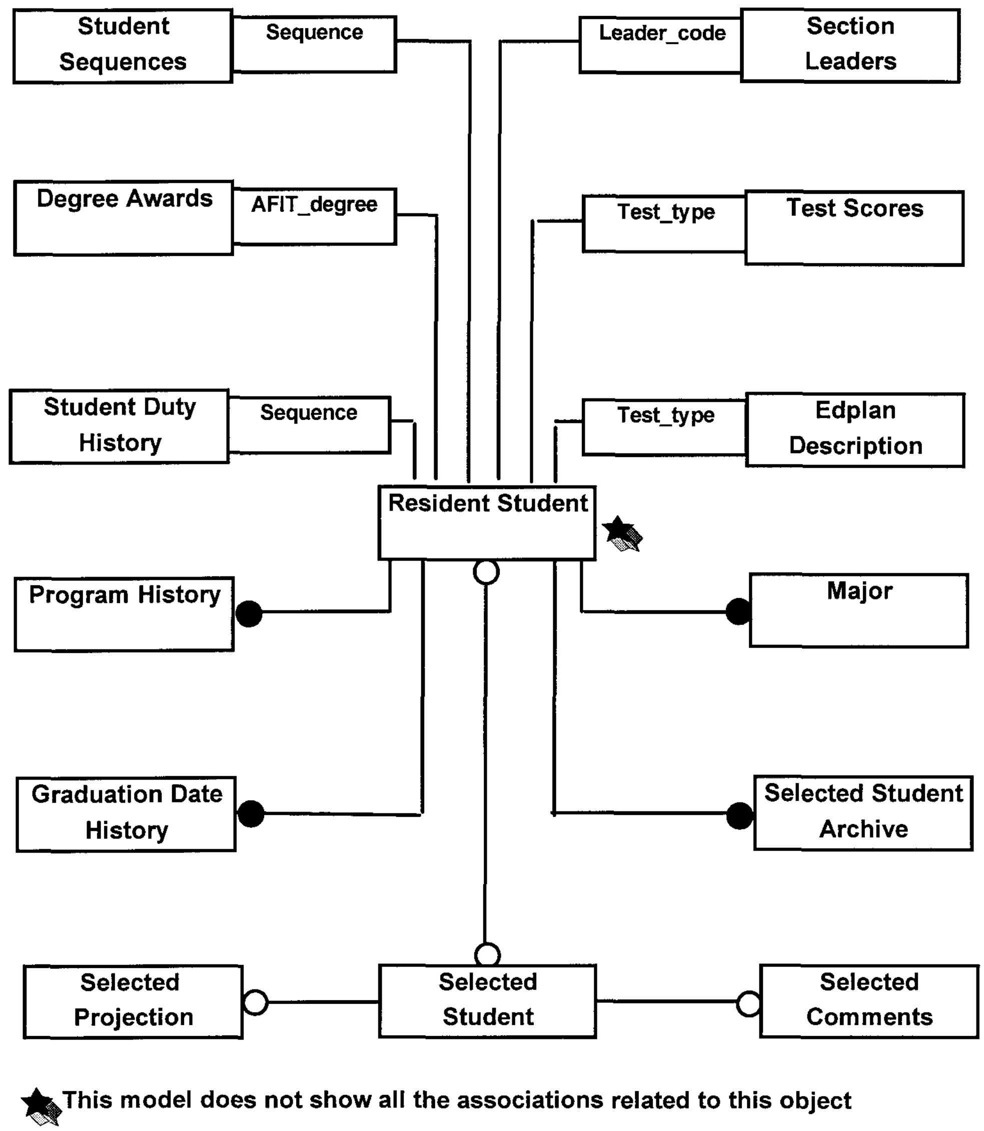


Figure 17: Resident Student Object Model

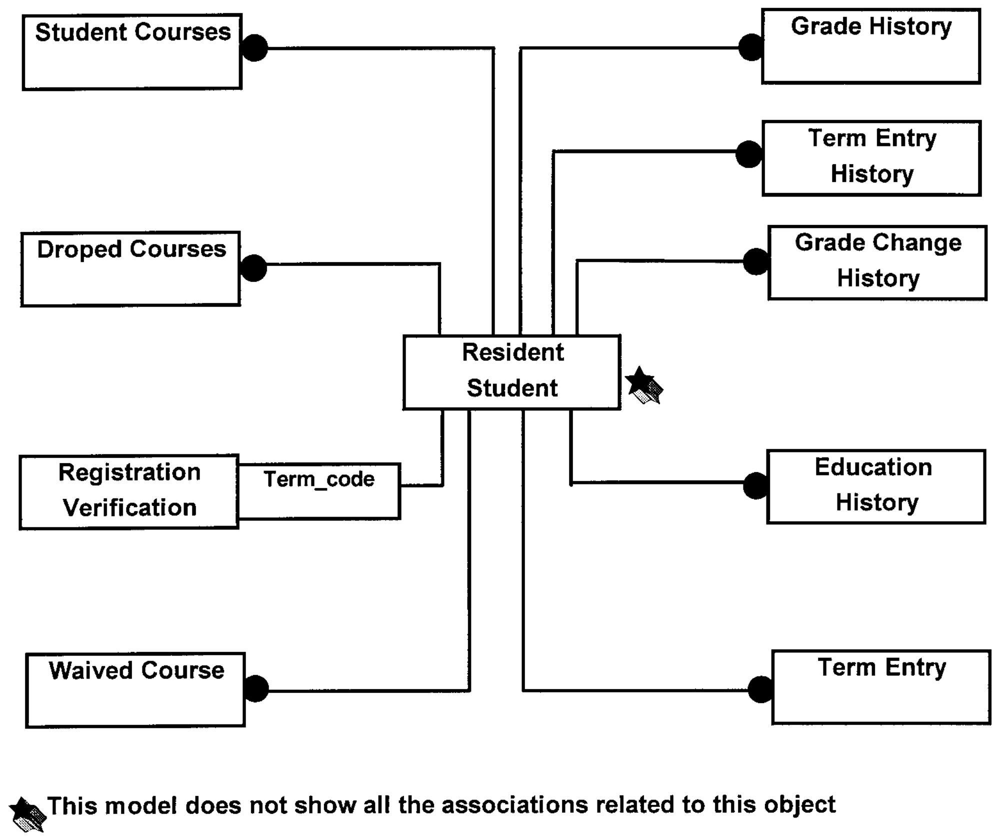


Figure 18: Resident Student Object Model (cont.)

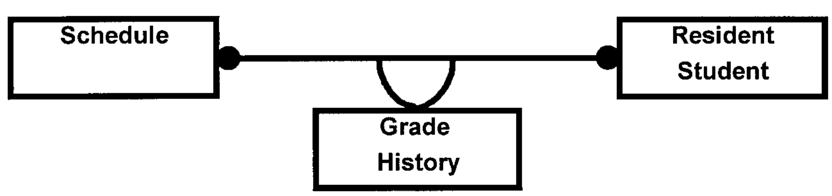


Figure 19: Grade History Association

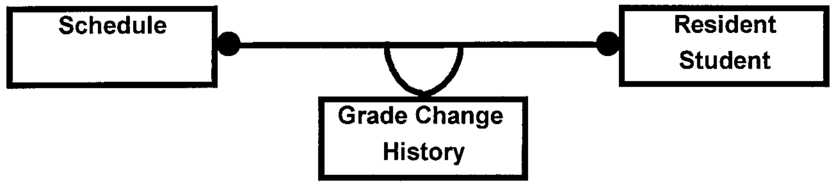


Figure 20: Grade Change History Association

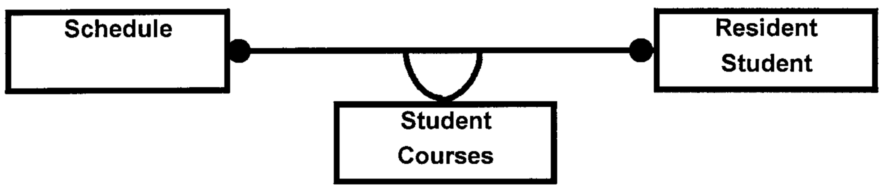


Figure 21: Student Courses Association

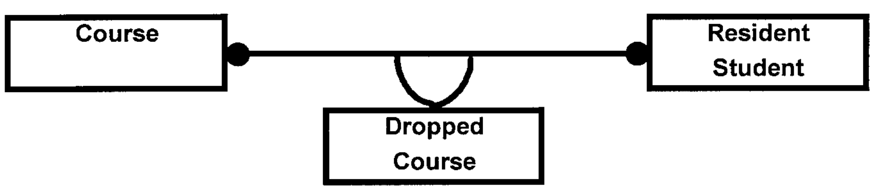


Figure 22: Dropped Course Association

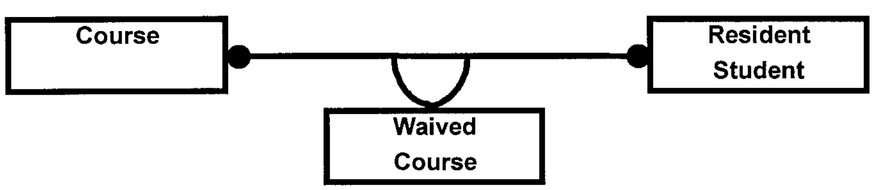


Figure 23: Waived Course Association

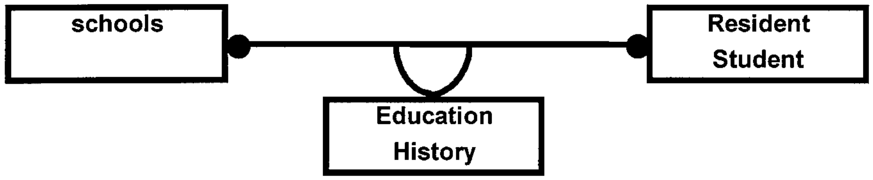


Figure 24: Education History Association

Person Structure Definition

Object Name: Person
Object Number:
Object Description: General model of a person

Author: Maj Pedro Arthur Linhares Lima
Date: 03/25/96
History: Thesis

Superclass: None
Components: None
Context: None

Attributes:

SSAN	SSAN type	Social Security Account Number.
lastname	String	Person's last name.
firstname	String	Person's first name.
name_prefix	Prefix type	Name prefix.
name_suffix	Suffix type	Name suffix.
middle_initial	Character	Name middle initial.
gender	{male, female}	
birth_date	Date type	Birth date.
marital_status	{single, married, divorced, widow}	
race	Race type	Person's race.
religion	Religion type	Person's religion.
ethnic_group	Ethnic type	Person's ethnicity.
badge_number	Badge number type	Badge number.
email_address	String	Electronic mail address.
academic_ed_status	Academic_ed type	Academic education status.
duty_title	String	Duty title.
duty_phone	Phone type	Duty phone number.
department_code	Department type	Person's department.
citizenship_country	Country type	Person's country.
login_name	String	Login name of who made changes.
input_date	Date type	Date of last change.

Constraints:

Z Static Schema:

Let **SSAN_TYPE** be the set of all Social Security Account Numbers.
Let **DATE_TYPE** be the set of all possible dates.
Let **PREFIX_TYPE** be the set of all possible name prefixes.
Let **SUFFIX_TYPE** be the set of all possible name suffixes.
Let **RACE_TYPE** be the set of all possible races.
Let **RELIGION_TYPE** be the set of all possible religions.
Let **ETHNIC_TYPE** be the set of all possible ethnic groups.
Let **BADGE_NUMBER_TYPE** be the set of all possible badge numbers.
Let **ACADEMIC_ED_TYPE** be the set of all possible academic education types.
Let **PHONE_TYPE** be the set of all possible phone numbers.
Let **DEPARTMENT_TYPE** be the set of all possible departments.
Let **COUNTRY_TYPE** be the set of all possible country codes.

Person

SSAN : SSAN_TYPE
lastname : String
firstname : String
name_prefix : PREFIX_TYPE
name_suffix : SUFFIX_TYPE
middle_initial : Character
gender : {male, female}
birth_date : DATE_TYPE
race : RACE_TYPE
marital_status : {single, married, divorced, widow}
religion : RELIGION_TYPE
email_address : String
academic_ed_status : ACADEMIC_ED_TYPE
ethnic_group : ETHNIC_TYPE
citizenship_country : COUNTRY_TYPE
department_code : DEPARTMENT_TYPE
duty_title : String
duty_phone :PHONE_TYPE
badge_number : BADGE_TYPE
login_name : String
input_date : DATE_TYPE

Military Structure Definition

Object Name: Military
Object Number:
Object Description: General model of a military

Author: Maj Pedro Arthur Linhares Lima
Date: 03/27/96
History: Thesis

Superclass: Person
Components: None
Context: None

Attributes:

rank	Rank type	Military rank.
branch	Branch type	Branch of service.
date_of_rank	Date type	Date of rank.
AFSC	AFSC type	AFSC code.
date_of_commission	Date type	Date of commission.
date_of_separation	Date type	Date of separation.
manning_code	Manning type	Manning code.
DEROS_date	Date type	DEROS date.
duty_effective_date	Date type	Date of effective duty.
aero_rating_code	Aero_rating type	Aero rating code.
MAJCOM	MAJCOM type	MAJCOM code.
base	Base type	Base code.
blue_chip_indicator	Character	Blue chip indicator.
NCO_indicator	Boolean	NCO indicator.
MPC_code	MPC type	MPC code.
recall_roster	Person pointer	Pointer to another military.

Constraints:

Z Static Schema:

Let **RANK_TYPE** be the set of all possible rank types.
Let **BRANCH_TYPE** be the set of all possible branch types.
Let **DATE_TYPE** be the set of all possible dates.
Let **MANNING_TYPE** be the set of all possible manning types.
Let **AERO_RATING_TYPE** be the set of all possible aero rating types.
Let **MAJCOM_TYPE** be the set of all possible MAJCOM types.
Let **BASE_TYPE** be the set of all possible base types.
Let **MPC_TYPE** be the set of all possible MPC types.
Let **PERSON_POINTER_TYPE** be a pointer to a particular person.

Military

rank : RANK_TYPE
branch : BRANCH_TYPE
date_of_rank : DATE_TYPE
AFSC : AFSC_TYPE
date_of_commission : DATE_TYPE
date_of_separation : DATE_TYPE
manning_code : MANNING_TYPE
DEROS_date : DATE_TYPE
duty_effective_date : DATE_TYPE
aero_rating_code : AERO_RATING_TYPE
MAJCOM : MAJCOM_TYPE
base : BASE_TYPE
blue_chip_indicator : Character
NCO_indicator : Boolean
MPC_code : MPC_TYPE
recall_roster : PERSON_POINTER_TYPE

Student Structure Definition

Object Name: Student
Object Number:
Object Description: General model of student

Author: Maj Pedro Arthur Linhares Lima
Date: 03/27/96
History: Thesis

Superclass: Person
Components: None
Context: None

Attributes:

Part_Record_Indicator	Boolean	If a student is resident or part time.
Ed_Level	Education_level type	Major education level.
Selected_Type	Selected type	School that has been selected.
Gaining_AFSC	AFSC type	The AFSC that he is going.
Gain_MAJCOM	MAJCOM type	The MAJCOM that he is going.
Gain_duty_Station	Duty_Station type	Duty station he is going.
Losing_MAJCOM	MAJCOM type	The MAJCOM that he is losing.
PSE	PSE type	The professional specialized educ.
Last_Year_Attended	Year type	The last year he attended a schooll.

Constraints:

Z Static Schema:

Let **EDUCATION_LEVEL_TYPE** be the set of all education level types.
Let **SELECTED_TYPE** be the set of all possible selected types.
Let **AFSC_TYPE** be the set of all possible AFSC types.
Let **MAJCOM_TYPE** be the set of all possible MAJCOM types.
Let **DUTY_STATION_TYPE** be the set of all possible duty stations.
Let **PSE_TYPE** be the set of all possible PSE types.
Let **YEAR_TYPE** be the set of all possible years.

Student

Part_Record_Indicator : Boolean
Ed_Level : EDUCATION_LEVEL_TYPE
Selected_Type : SELECTED_TYPE
Gaining_AFSC : AFSC_TYPE
Gain_MAJCOM : MAJCOM_TYPE
Gain_duty_Station : DUTY_STATION_TYPE
Losing_MAJCOM : MAJCOM_TYPE
PSE : PSE_TYPE
Last_Year_Attended : YEAR_TYPE

Resident Student Structure Definition

Object Name: Resident Student
Object Number:
Object Description: General model of resident student

Author: Maj Pedro Arthur Linhares Lima
Date: 03/27/96
History: Thesis

Superclass: Student
Components: None
Context: None

Attributes:

Academic_Action	Academic_Action type	Student's academic standing.
Overdue_Indicator	Boolean	If a student has an overdue book.
Classification	Classification type	Represents an enrollment classific.
Major_ASC	ASC type	Major Academic specialty.
Program	Program type	Student's Program.
Class	Class type	Student's class.
Program_Year	Year type	Prefix of the program year.
AFIT_Degree	Degree type	Type of AFIT degree.
Career_Pointer_Code	Career pointer type	Level of education that credit apply
Departure_Date	Date type	Departure date from AFIT.
Box_Number	Box type	Student's box number.
Card_Number	Card type	Student's card number.
Library_Number	Library_Number type	Student's library number.
Locker_Number	Locker type	Student's locker number.
Student_Sponsor	Person pointer	Pointer to sponsor.
Faculty_Advisor	Person pointer	Pointer to faculty advisor.

Constraints:

Z Static Schema:

Let **ACADEMIC_ACTION _TYPE** be the set of all possible academic actions.
Let **CLASSIFICATION_TYPE** be the set of all possible classification types.
Let **ASC_TYPE** be the set of all possible ASC types.
Let **PROGRAM_TYPE** be the set of all possible programs.
Let **CLASS_TYPE** be the set of all possible class types.
Let **YEAR_TYPE** be the set of all possible years.
Let **DEGREE_TYPE** be the set of all possible degree types.
Let **CAREER_POINTER_TYPE** be the set of all possible career pointer types.
Let **DATE_TYPE** be the set of all possible dates.
Let **BOX_TYPE** be the set of all possible boxes.
Let **CARD_POINTER_TYPE** be the set of all possible card pointer types.
Let **LIBRARY_NUMBER_TYPE** be the set of all possible library numbers.
Let **LOCKER_TYPE** be the set of all possible locker numbers.
Let **PSE_TYPE** be the set of all possible PSE types.
Let **PERSON_POINTER_TYPE** be a pointer to a particular person.

Resident student

Academic_Action : ACADEMIC_ACTION_TYPE
Overdue_Indicator : Boolean
Classification : CLASSIFICATION_TYPE
Major_ASC : ASC_TYPE
Program : PROGRAM_TYPE
Class : CLASS_TYPE
Program_Year : YEAR_TYPE
AFIT_Degree : DEGREE TYPE
Career_Pointer_Code : CAREER_POINTER_TYPE
Departure_Date : DATE_TYPE
Box_Number : BOX_TYPE
Card_Number : CARD_TYPE
Library_Number : LIBRARY_NUMBER_TYPE
Locker_Number : LOCKER_TYPE
Student_Sponsor : PERSON_POINTER_TYPE
Faculty_Advisor : PERSON_POINTER_TYPE

INTL-Student Structure Definition

Object Name: INTL-Student
Object Number:
Object Description: General model of INTL-student

Author: Maj Pedro Arthur Linhares Lima
Date: 03/27/96
History: Thesis

Superclass: Resident Student
Components: None
Context: None

Attributes:

WSCN	WSCN type	Work Sheet Control Number.
ITO	ITO type	ITO number.
Case_number	Case_number type	Case number.
DLI_request_indicator	Boolean	If student has to attended DLI.
DLI_indicator	Boolean	If student attended DLI.
evaluation_req_date	Date type	Date evaluation was requested.
requested_program	Program type	Requested student's Program.
eval_forward_date	Date type	Evaluation forward date.
forward_to_dept	Department type	Department it was forwarded.
Eval_returned_date	Date type	Date the evaluation returned.
admission_status	Admission type	Admission status code.
eval_remarks	String	Evaluation remarks.
country_notified_date	Date type	Date the country was notified.
AFSAT_notified_date	Date type	Date that AFSAT was notified.
AFSAT_quota_indicator	Boolean	If student fills a country's quota.
first_sponsor	Person pointer	Pointer to sponsor.
second_sponsor	Person pointer	Pointer to sponsor.
source_of_funds	Funds type	Source of funds code.
AFSAT_country	Country type	The home country of a student.

Constraints:

Z Static Schema:

Let **WSCN _TYPE** be the set of all possible WSCN numbers.
Let **ITO_TYPE** be the set of all possible ITO numbers.
Let **CASE_NUMBER_TYPE** be the set of all possible case numbers.
Let **DATE_TYPE** be the set of all possible dates.
Let **PROGRAM_TYPE** be the set of all possible programs.
Let **DEPARTMENT_TYPE** be the set of all possible department types.
Let **ADMISSION_TYPE** be the set of all possible admission types.
Let **PERSON_POINTER_TYPE** be a pointer to a particular person.
Let **FUNDS_TYPE** be the set of all possible funds types.
Let **COUNTRY_TYPE** be the set of all possible country types.

INTL-student

WSCN : WSCN_TYPE
ITO : ITO_TYPE
Case_number : CASE_NUMBER_TYPE
DLI_request_indicator : Boolean
DLI_indicator : Boolean
evaluation_req_date : DATE_TYPE
requested_program : PROGRAM_TYPE
eval_forward_date : DATE_TYPE
forward_to_dept : DEPARTMENT_TYPE
Eval_returned_date : DATE_TYPE
admission_status : ADMISSION_TYPE
eval_remarks : String
country_notified_date : DATE_TYPE
AFSAT_notified_date : DATE_TYPE
AFSAT_quota_indicator : Boolean
first_sponsor : PERSON_POINTER_TYPE
second_sponsor : PERSON_POINTER_TYPE
source_of_funds : FUNDS_TYPE
AFSAT_country : COUNTRY_TYPE

Address Structure Definition

Object Name: Address
Object Number:
Object Description: General model of address

Author: Maj Pedro Arthur Linhares Lima
Date: 03/25/96
History: Thesis

Superclass: None
Components: None
Context: None

Attributes:

address_type	Address type	Address type.
address	String	Address.
city	String	City.
state	State type	State.
country	Country type	Country.
zipcode	Zip type	Zip code.
phone	Phone type	Phone number.
address_effective_date	Date type	Date of effective address.
login_name	String	Login name of who made changes.
login_date	Date type	Date of last change.

Constraints:

Z Static Schema:

Let **ADDRESS_TYPE** be the set of all possible address types.
Let **STATE_TYPE** be the set of all possible states.
Let **COUNTRY_TYPE** be the set of all possible country codes.
Let **ZIP_TYPE** be the set of all possible zip codes.
Let **PHONE_TYPE** be the set of all phone Numbers.
Let **DATE_TYPE** be the set of all possible dates.

Address

address_type : String
address : String
city : String
state : STATE_TYPE
zipcode : ZIP_TYPE
country : COUNTRY_TYPE
phone : PHONE_TYPE
address_effective_date : DATE_TYPE
login_name : String
login_date : DATE_TYPE

Dependent Information Structure Definition

Object Name: Dependent Information
Object Number:
Object Description: General model of dependent information

Author: Maj Pedro Arthur Linhares Lima
Date: 03/26/96
History: Thesis

Superclass: None
Components: None
Context: None

Attributes:

number_children	Integer	Number of student's children.
single_dep_chldrn	Character	Indicator if single with children.
deps_at_AFIT	Character	Indicator if dependents at AFIT.

Constraints:

Z Static Schema:

Dependent Information

number_children : Integer
single_dep_chldrn: Character
deps_at_AFIT: Character

Education History Structure Definition

Object Name: Education History
Object Number:
Object Description: General model of education history

Author: Maj Pedro Arthur Linhares Lima
Date: 03/26/96
History: Thesis

Superclass: None
Components: None
Context: None

Attributes:

MPC_School_Code	MPC type	Military Personnel Center school.
Ed_Level_Code	Ed_level type	Education level.
Type_Degree_Code	Type_degree type	Type of degree.
ASC_Code	ASC type	Academic specialty code.
Quality_Points	Integer	Quality points.
Total_Credit_Hours	Integer	Total of credit hours.
Method_Of_Obtainment	Method_of_obt type	Method of obtained an education.
Academic_Ed_Status	Academic_ed type	Academic education status.
Last_Year_Attended	Date type	Last year that attended a school.
ABET_Accredited	Boolean	If AFIT degree's ABET accredited.
Ed_History_Remarks	String	Education history remarks.
Work_ID_Processed	Work_ID type	Action that orig. a transaction.
Login_Name	String	Login name of who made changes.
Input_Date	Date type	Date of last change.
Operators_Initials	String	Operator's initials.
Trnscrpt_Career_Pointer	Transcript type	Education level of student transcr.
Duty_Location	Duty_location type	Location of active duty.
Degree_Cum_Gpa	Integer	The cumulative student's GPA.
Degree_Title	Degree type	The title of a student's degree.

Constraints:

Z Static Schema:

Let **MPC_TYPE** be the set of all possible MPC types.
Let **ED_LEVEL_TYPE** be the set of all possible education level types.
Let **TYPE_DEGREE_TYPE** be the set of all possible degree types.
Let **ASC_TYPE** be the set of all possible ASC types.
Let **DATE_TYPE** be the set of all possible dates.
Let **METHOD_OF_OBT_TYPE** be the set of all possible method of obtained types.
Let **ACADEMIC_ED_TYPE** be the set of all possible academic education types.
Let **WORK_ID_TYPE** be the set of all possible work ID types.
Let **TRANSCRIPT_TYPE** be the set of all possible transcript types.
Let **DUTY_LOCATION_TYPE** be the set of all possible duty location types.
Let **DEGREE_TYPE** be the set of all possible degree types.

Education History

MPC_School_Code : MPC_TYPE
Ed_Level_Code : ED_LEVEL_TYPE
Type_Degree_Code : TYPE_DEGREE_TYPE
ASC_Code : ASC_TYPE
Quality_Points : Integer
Total_Credit_Hours : Integer
Method_Of_Obtainment :METHOD_OF_OBT_TYPE
Academic_Ed_Status : ACADEMIC_ED_TYPE
Last_Year_Attended : DATE_TYPE
ABET_Accredited : Boolean
Ed_History_Remarks : String
Work_ID_Processed : WORK_ID_TYPE
Login_Name : String
Input_Date : DATE_TYPE
Operators_Initials : String
Trnscrpt_Career_Pointer : TRANSCRIPT_TYPE
Duty_Location : DUTY_LOCATION_TYPE
Degree_Cum_Gpa : Integer
Degree_Title : DEGREE_TYPE

Emergency Data Structure Definition

Object Name: Emergency Data
Object Number:
Object Description: General model of emergency data

Author: Maj Pedro Arthur Linhares Lima
Date: 03/27/96
History: Thesis

Superclass: None
Components: None
Context: None

Attributes:

Contact_Fname	String	Emergency contact first name.
Contact_Lname	String	Emergency contact last name.
Relation	Relation type	The relationship with a person.
Address	String	Home address.
City	String	Home city.
State	State type	Home state.
Zipcode	Zip type	Home zip code.
Country	Country type	Person's country.
Phone	Phone type	Home phone number.
Firm_Name_Office	Office type	Person's office symbol.
Additional_Address	String	Additional address information.
Street_Address	String	Additional street information.
Address_Room_Type	Room type	Additional Room type.
Address_Room_Number	String	Additional room number.
Street_Type_Code	Street type	Additional street type.
Revision_Name	String	Name of who made the revision.
Revision_Date	Date type	Revision date.
Login_Name	String	Login name of who made changes.
Login_Date	Date type	Date of last change.

Constraints:

Z Static Schema:

Let **RELATION_TYPE** be the set of all possible relationship types.
Let **STATE_TYPE** be the set of all possible states.
Let **ZIP_TYPE** be the set of all possible zip codes.
Let **COUNTRY_TYPE** be the set of all possible country codes.
Let **PHONE_TYPE** be the set of all phone numbers.
Let **OFFICE_TYPE** be the set of all possible offices.
Let **ROOM_TYPE** be the set of all possible room types.
Let **STREET_TYPE** be the set of all possible street types.
Let **DATE_TYPE** be the set of all possible dates.

Emergency data

Contact_Fname : String
Contact_Lname : String
Relation : RELATION_TYPE
Address : String
City : String
State : STATE_TYPE
Zipcode : ZIP_TYPE
Country : COUNTRY_TYPE
Phone : PHONE_TYPE
Firm_Name_Officel : OFFICE_TYPE
Additional_Address : String
Street_Address : String
Address_Room_Type : ROOM_TYPE
Address_Room_Number : String
Street_Type_Code : STREET_TYPE
Revision_Name : String
Revision_Date : DATE_TYPE
Login_Name : String
Login_Date : DATE_TYPE

Spouse Information Structure Definition

Object Name: Spouse Information
Object Number:
Object Description: General model of spouse information

Author: Maj Pedro Arthur Linhares Lima
Date: 03/27/96
History: Thesis

Superclass: None
Components: None
Context: None

Attributes:

Birth_Date	Date type	Spouse birth date.
Fname	String	Spouse first name.
Lname	String	Spouse last name.
Spouse_At_AFIT	Boolean	If spouse came with him/her.
Nickname	String	Spouse nickname.
Spouse_In_Military	Boolean	If spouse in military service.
Occupation	String	Spouse occupation.
Remarks	String	Remarks.

Constraints:

Z Static Schema:

Let **DATE_TYPE** be the set of all possible dates.

Spouse Information

Birth_Date : DATE_TYPE
Fname : String
Lname : String
Spouse_At_AFIT : Boolean
Nickname : String
Spouse_In_Military : Boolean
Occupation : String
Remarks : String

Student Duty History Structure Definition

Object Name: Student Duty History
Object Number:
Object Description: General model of student duty history

Author: Maj Pedro Arthur Linhares Lima
Date: 03/27/96
History: Thesis

Superclass: None
Components: None
Context: None

Attributes:

Title	Title type	Job title.
AFSC	AFSC type	Duty AFSC.
Organization	String	Where the student works.
Duty_Station	Duty_Station type	Student's duty station.
Assigned_Date	Date type	Date the student was assigned.
Sequence_Number	Integer	Sequence number.
Login_Name	String	Login name.

Constraints:

Z Static Schema:

Let **TITLE_TYPE** be the set of all possible titles.
Let **AFSC_TYPE** be the set of all possible AFSC types.
Let **DUTY_STATION_TYPE** be the set of all possible duty stations.
Let **DATE_TYPE** be the set of all possible dates.

Student duty history

Title : TITLE_TYPE
AFSC : AFSC_TYPE
Organization : String
Duty_Station : DUTY_STATION_TYPE
Assigned_Date : DATE_TYPE
Sequence_Number : Integer
Login_Name : String

Test Scores Structure Definition

Object Name: Test Scores
Object Number:
Object Description: General model of test scores

Author: Maj Pedro Arthur Linhares Lima
Date: 03/27/96
History: Thesis

Superclass: None
Components: None
Context: None

Attributes:

Test_Type	Test type	Type of test.
Taken_Date	Date type	The date of the test.
Score	Integer	Score on the test.
Input_Date	Date type	The date of the input.
Login_Name	String	Login name.

Constraints:

Z Static Schema:

Let **TEST_TYPE** be the set of all possible test types.
Let **DATE_TYPE** be the set of all possible dates.

Test scores

Test_Type : TEST_TYPE
Taken_Date : DATE_TYPE
Score : Integer
Input_Date : DATE_TYPE
Login_Name : String

Appendix D: The Functional Model

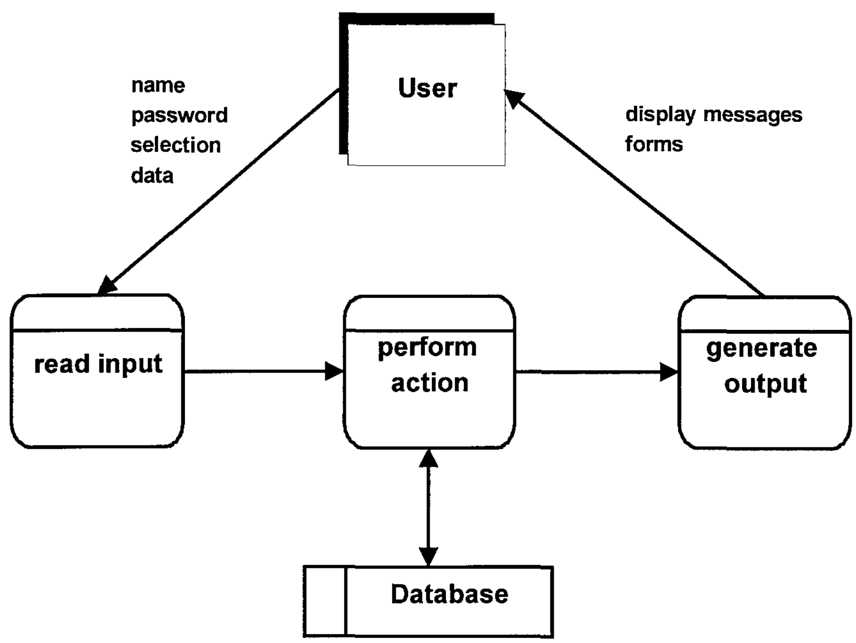


Figure 25: STARS Application Level 0 DFD

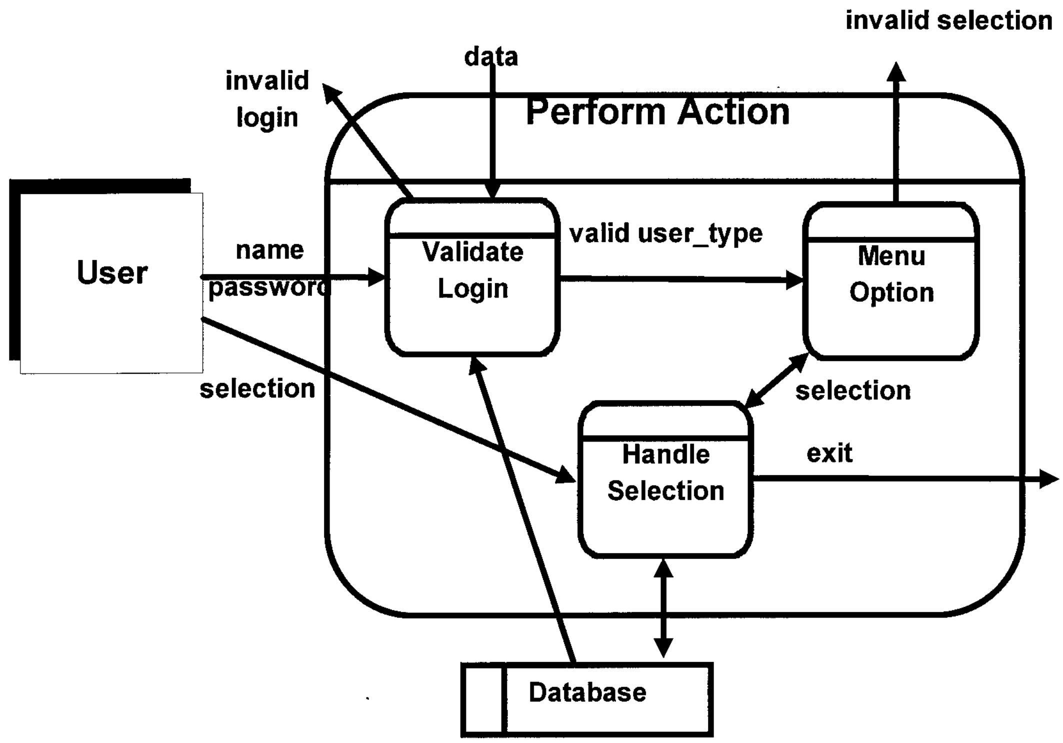


Figure 26: Perform Action Level 1 DFD

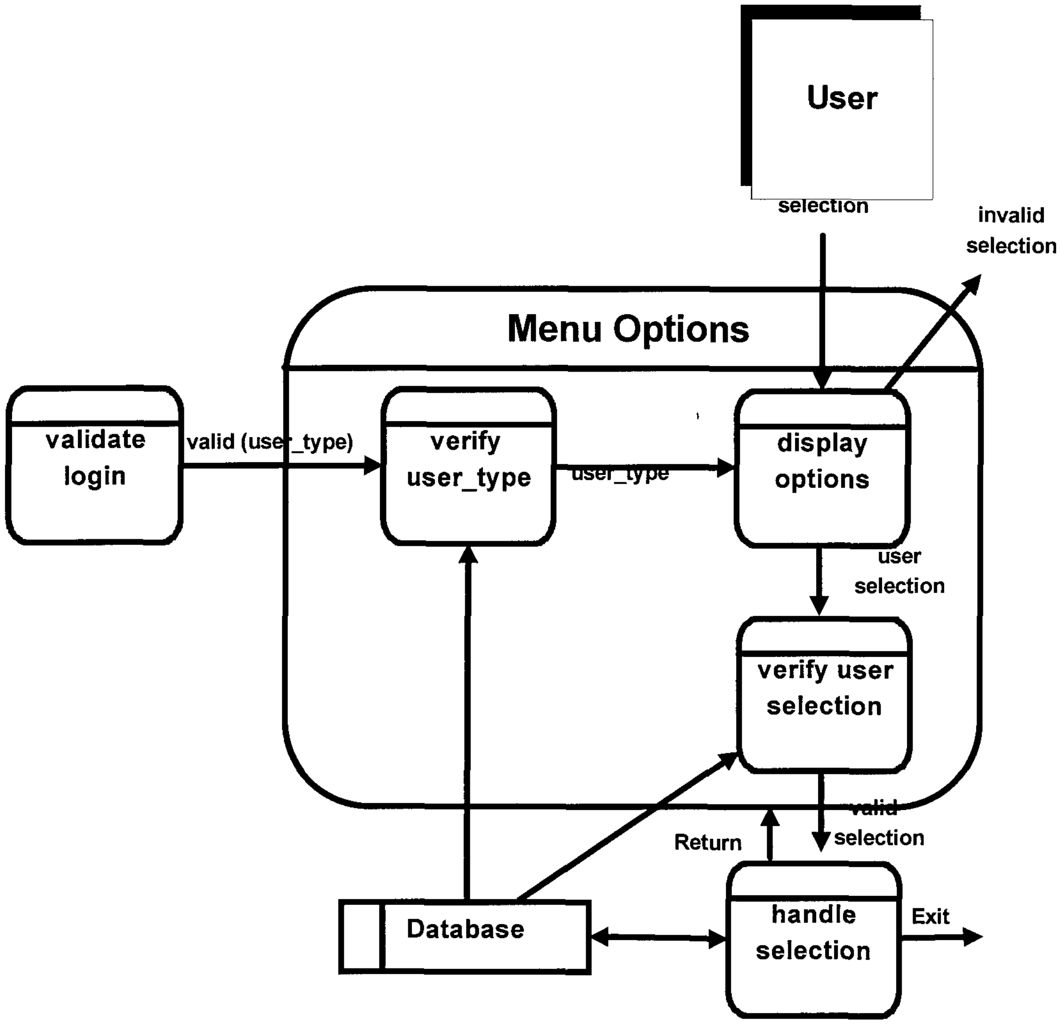


Figure 27: Menu Option Level 2 DFD

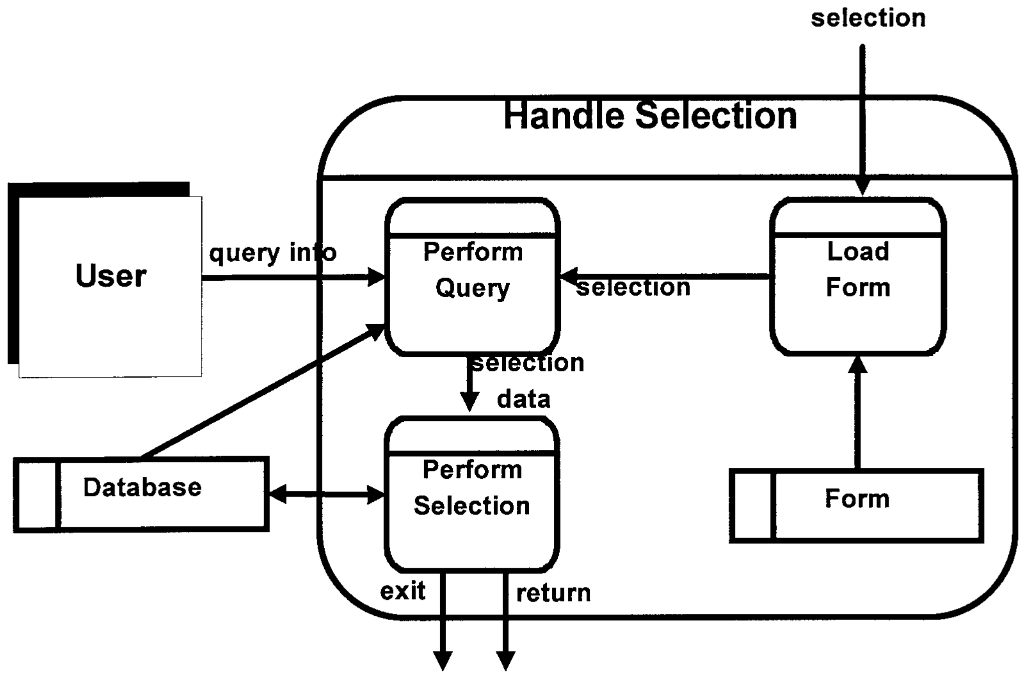


Figure 28: Handle Selection Level 2 DFD

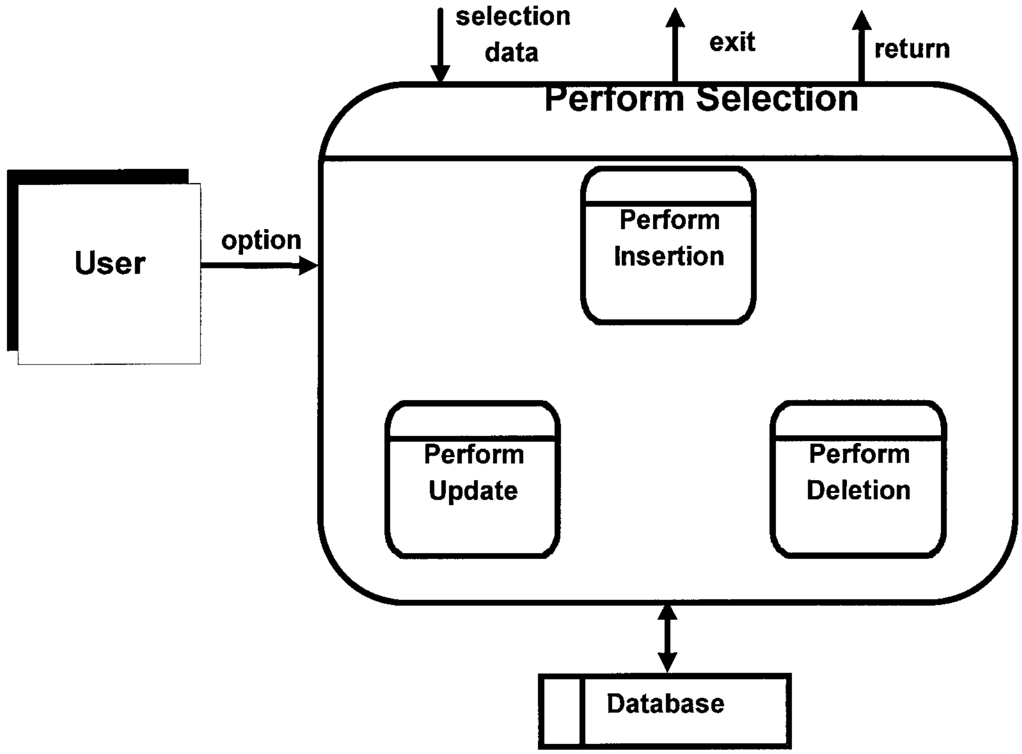


Figure 29: Perform Selection Level 3 DFD

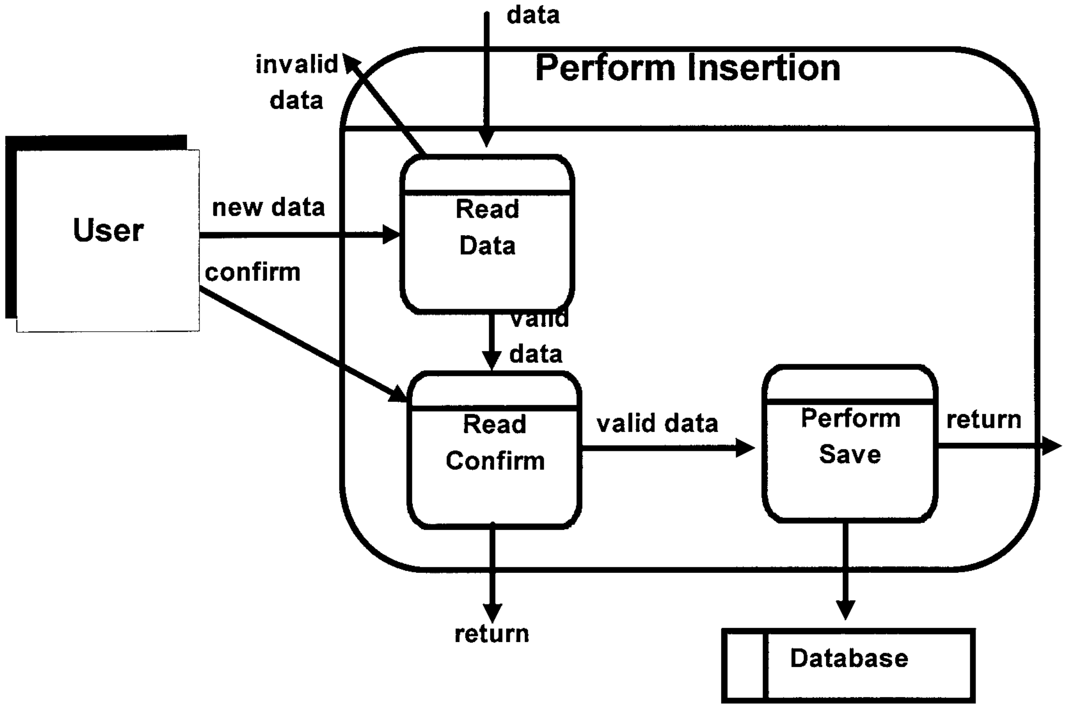


Figure 30: Perform Insertion Level 4 DFD

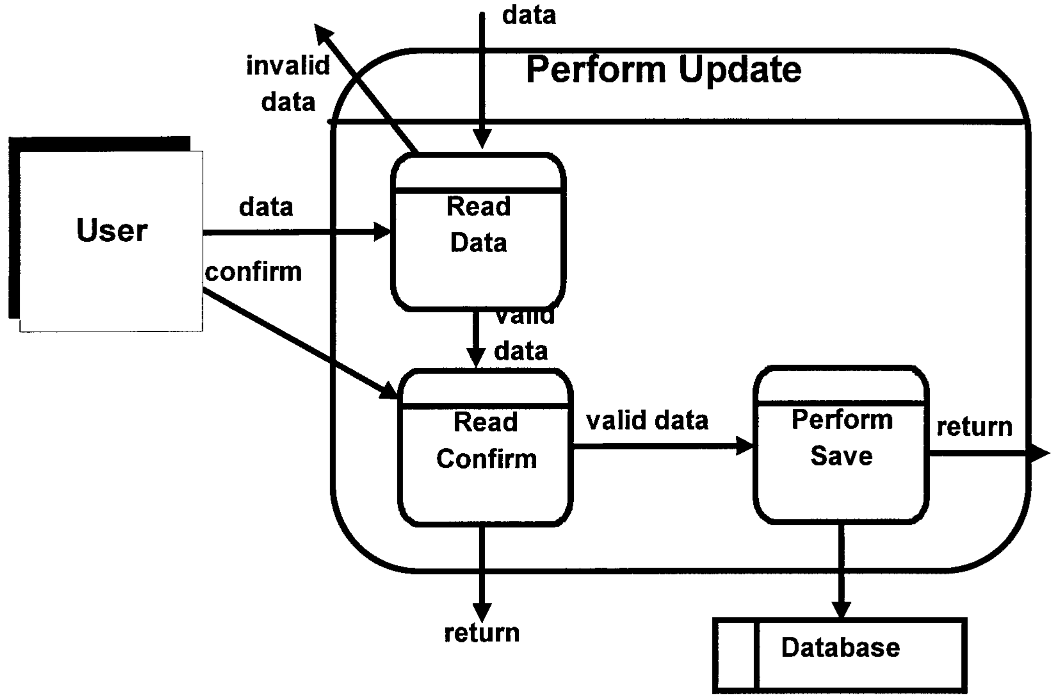


Figure 31: Perform Update Level 4 DFD

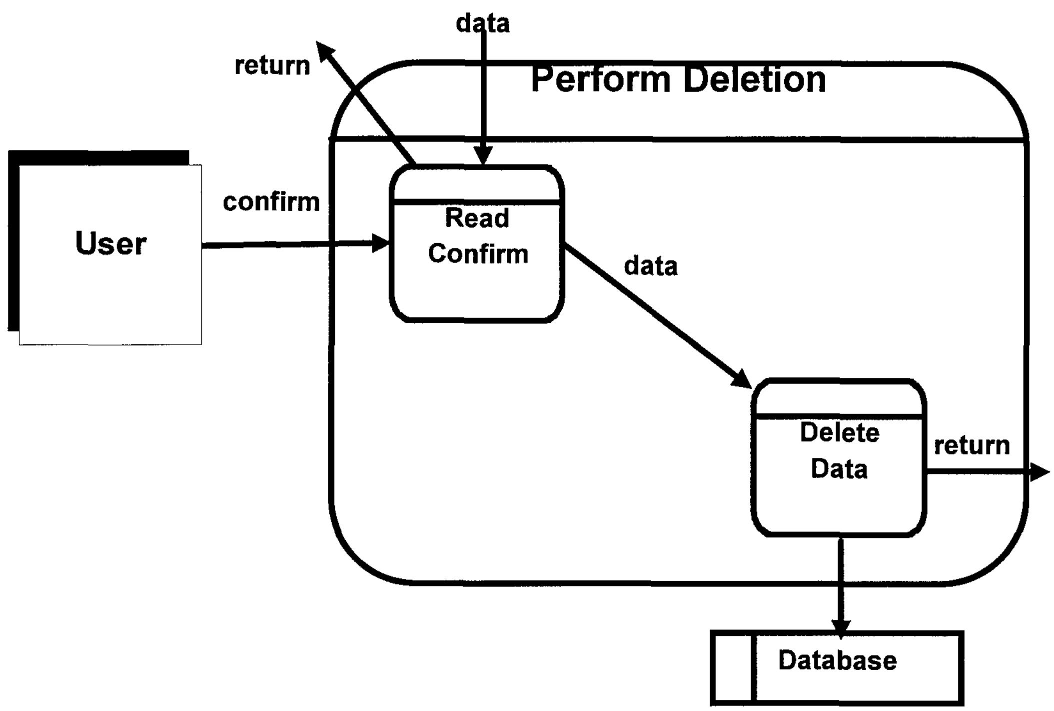


Figure 32: Perform Deletion Level 4 DFD

Appendix E: Implementation of the Object Model

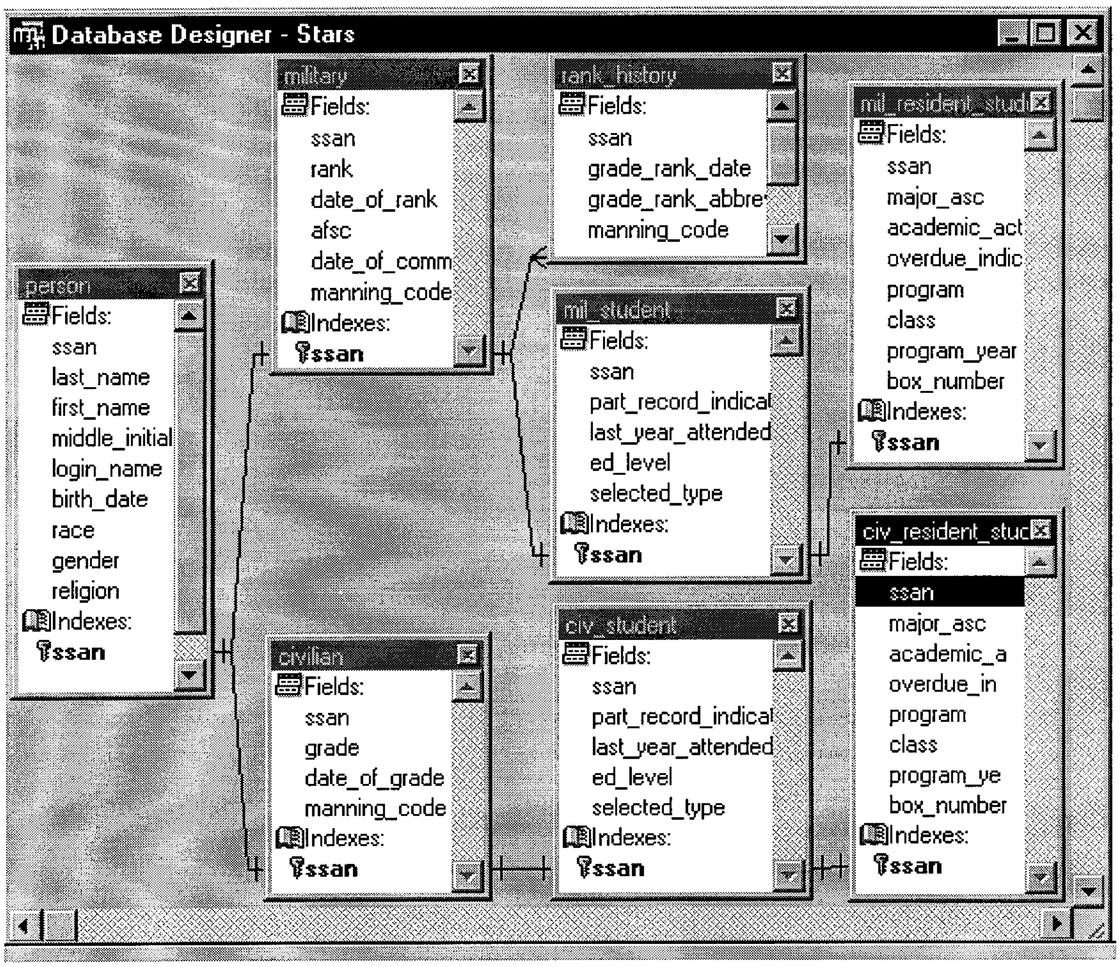


Figure 33: Table's relationship in STARS Database

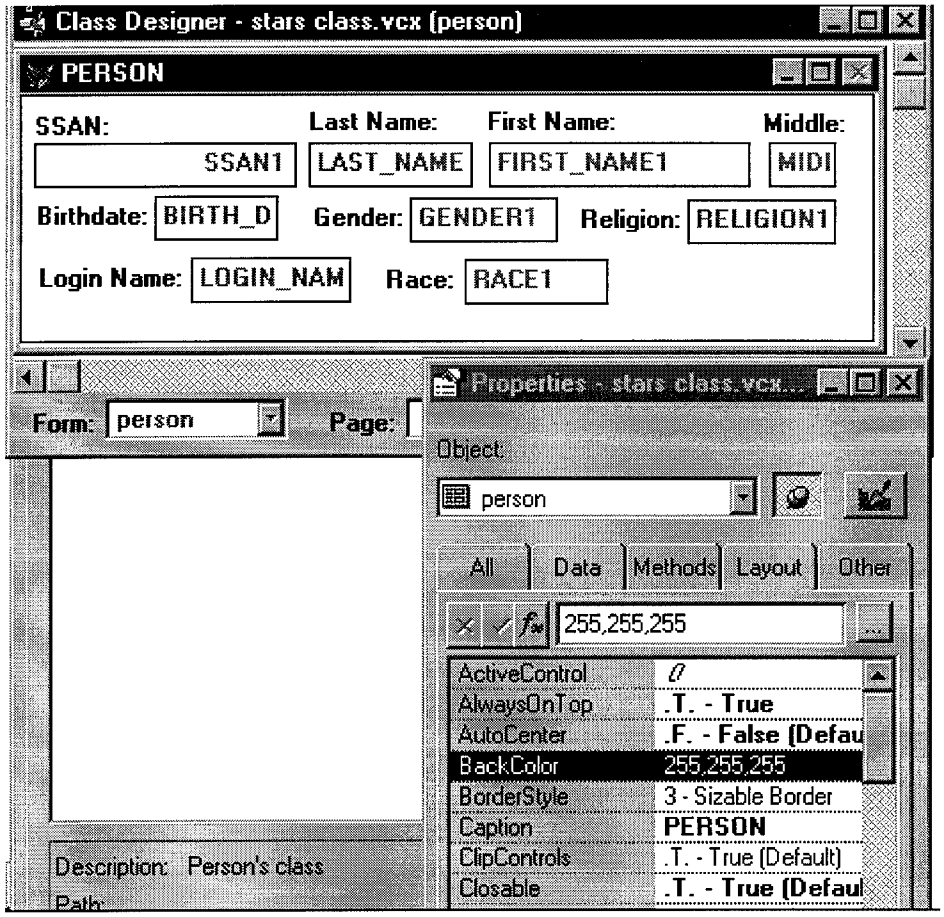
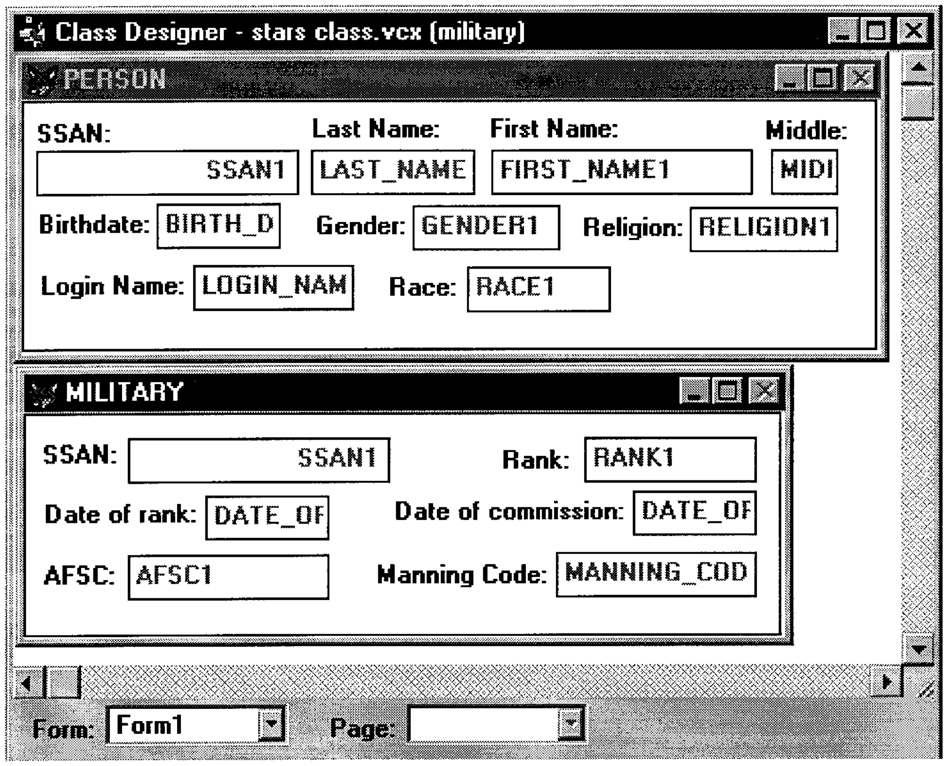


Figure 34: Person's Class



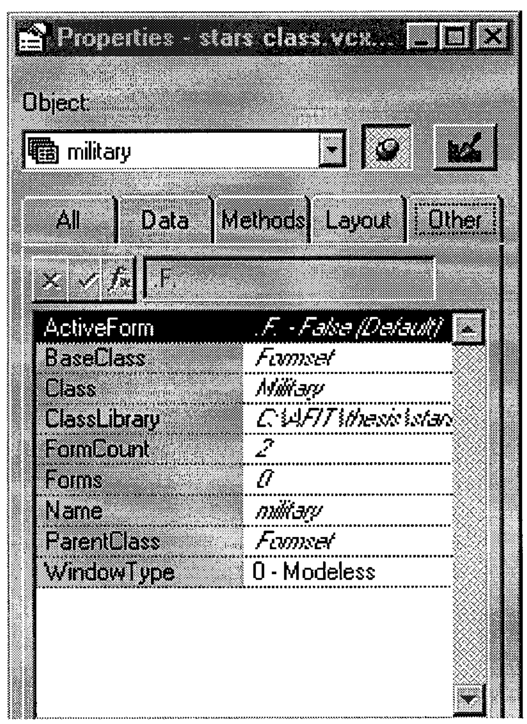


Figure 35: Military's Class

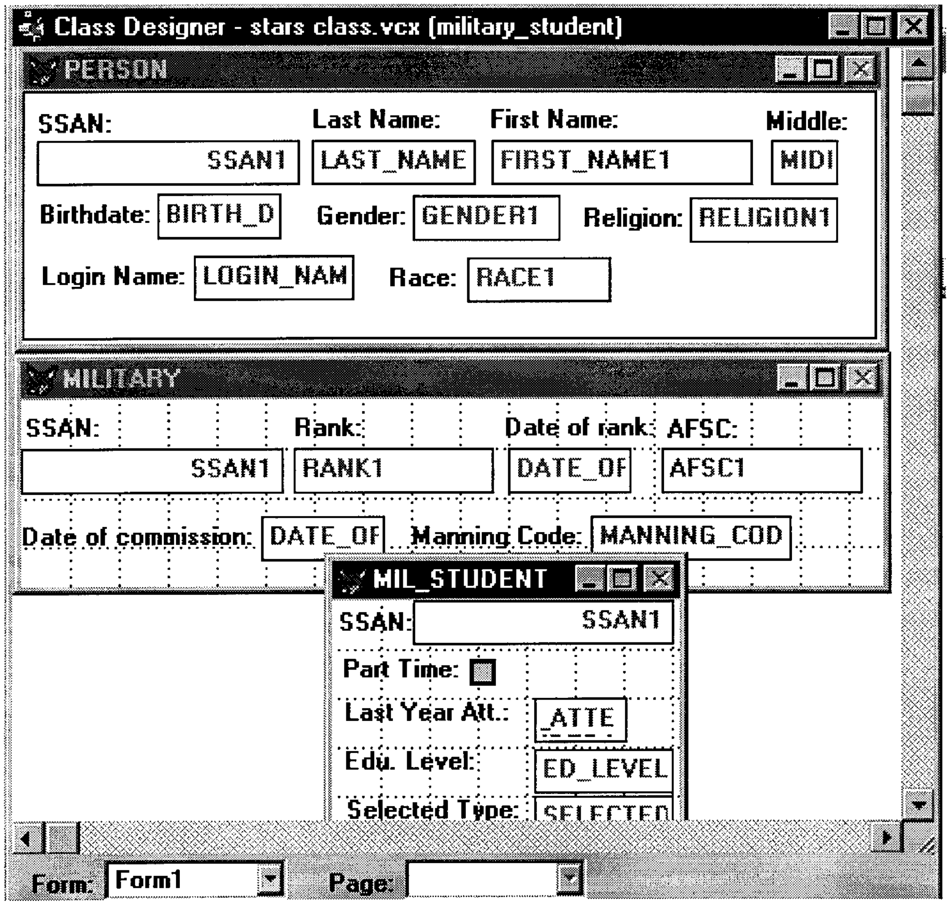


Figure 36: Military_student's Class

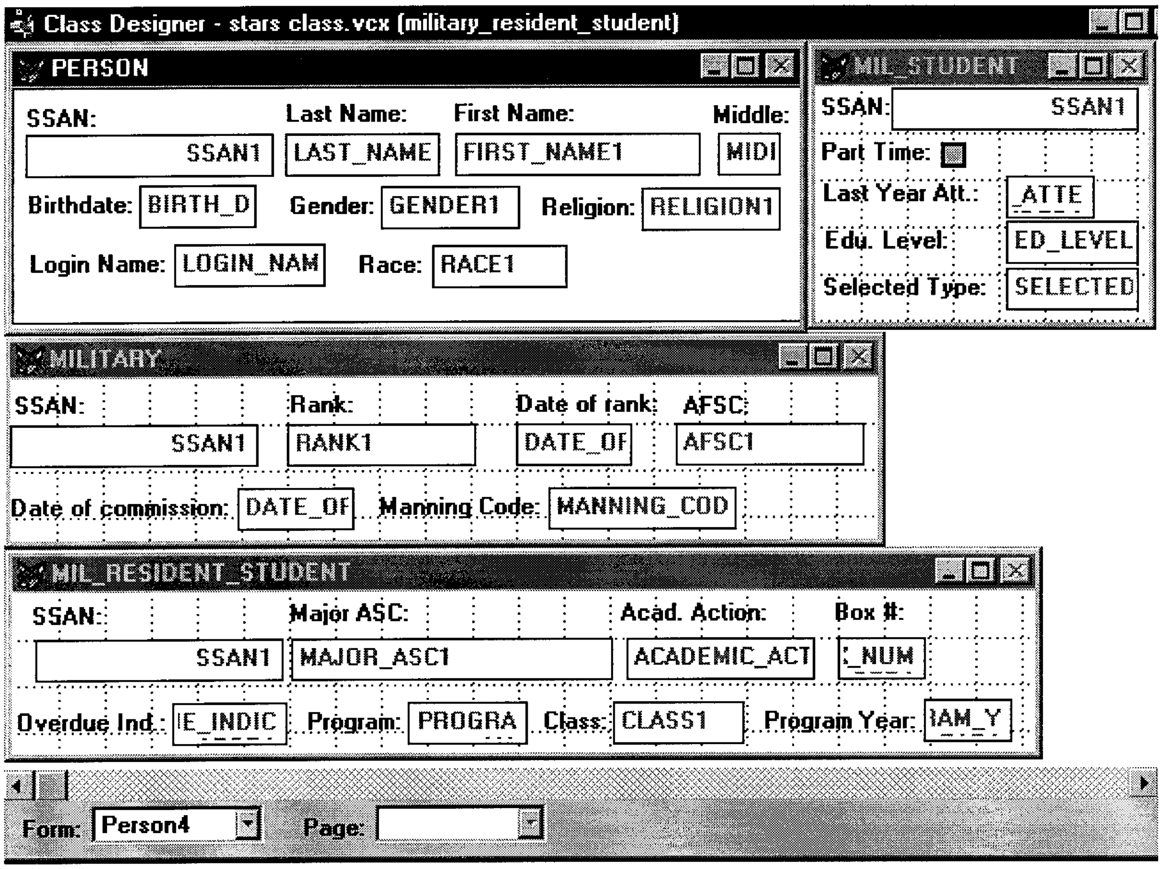


Figure 37: Military_resident_student's Class

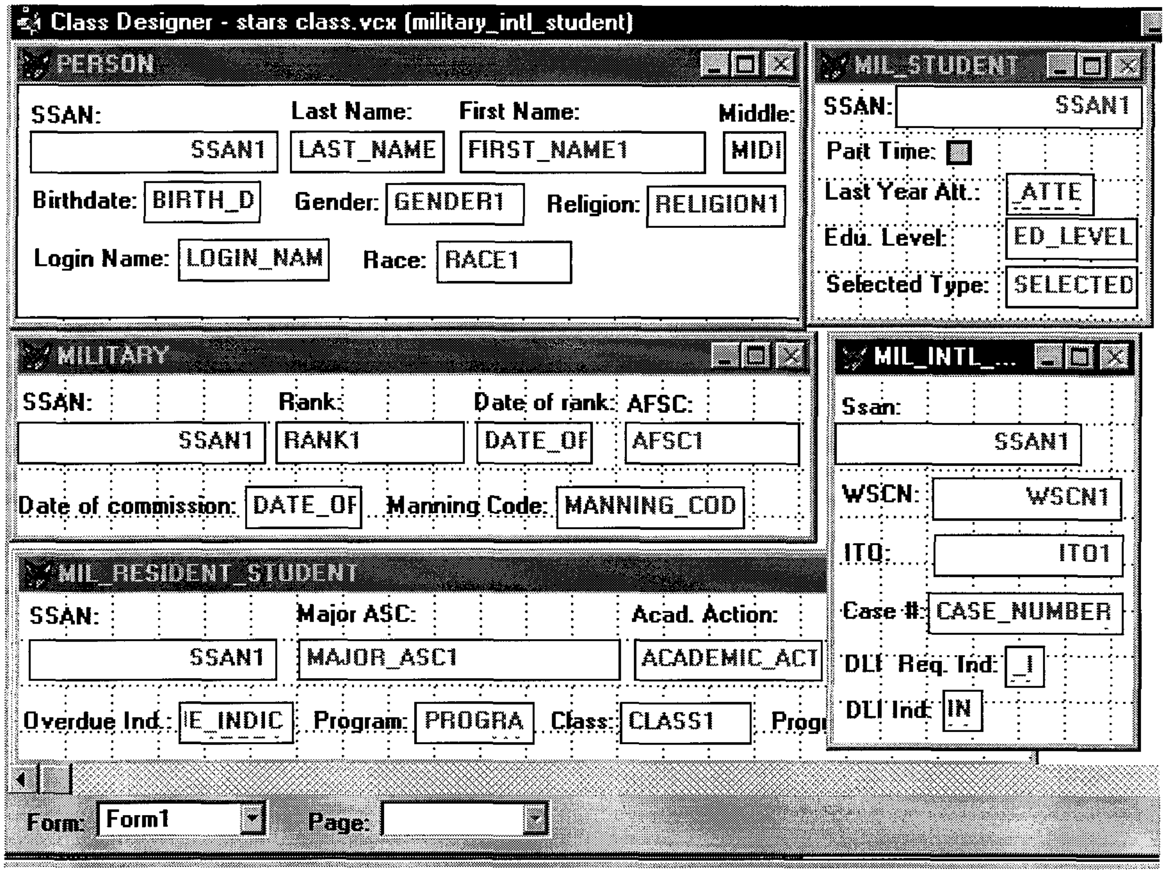


Figure 38: Military_INTL_student's Class

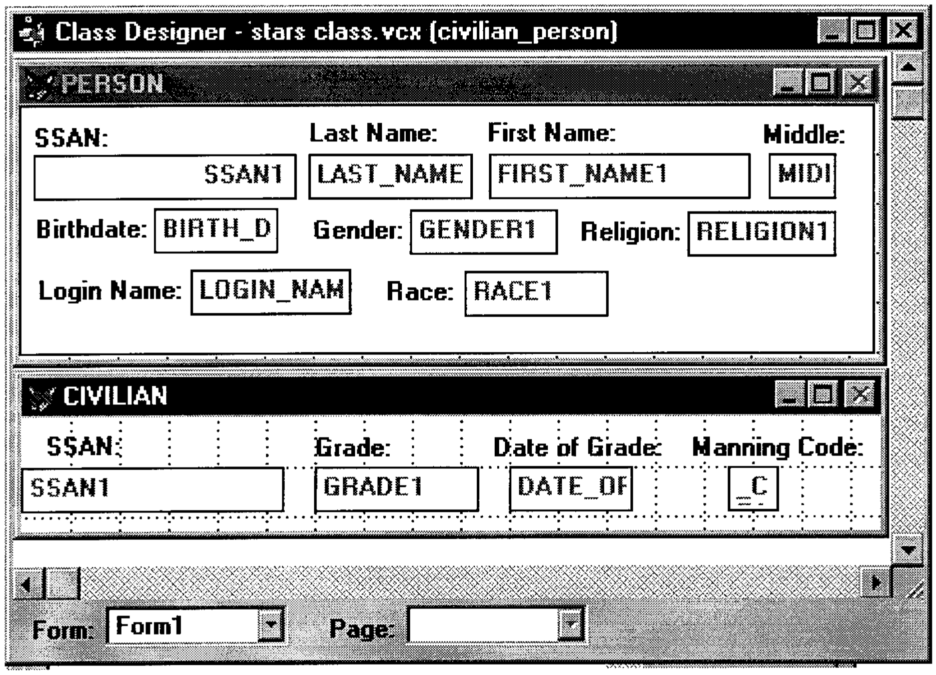


Figure 39: Civilian's Class

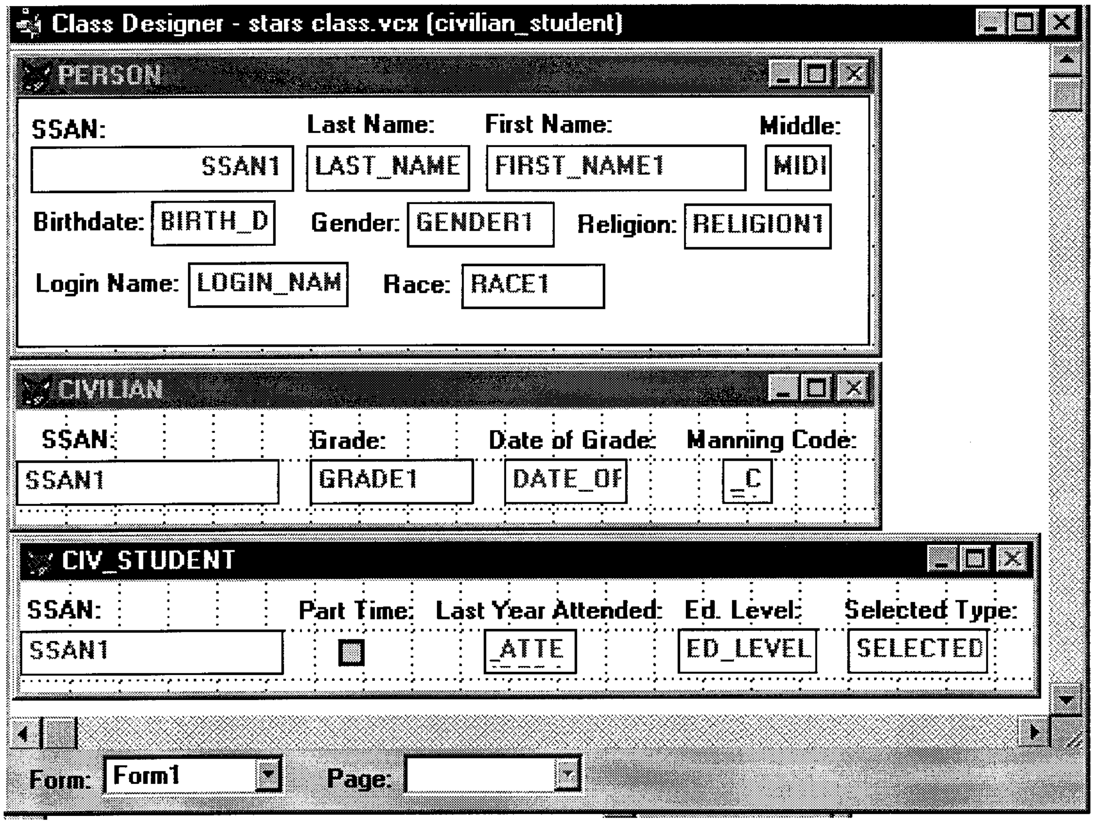


Figure 40: Civilian_student's Class

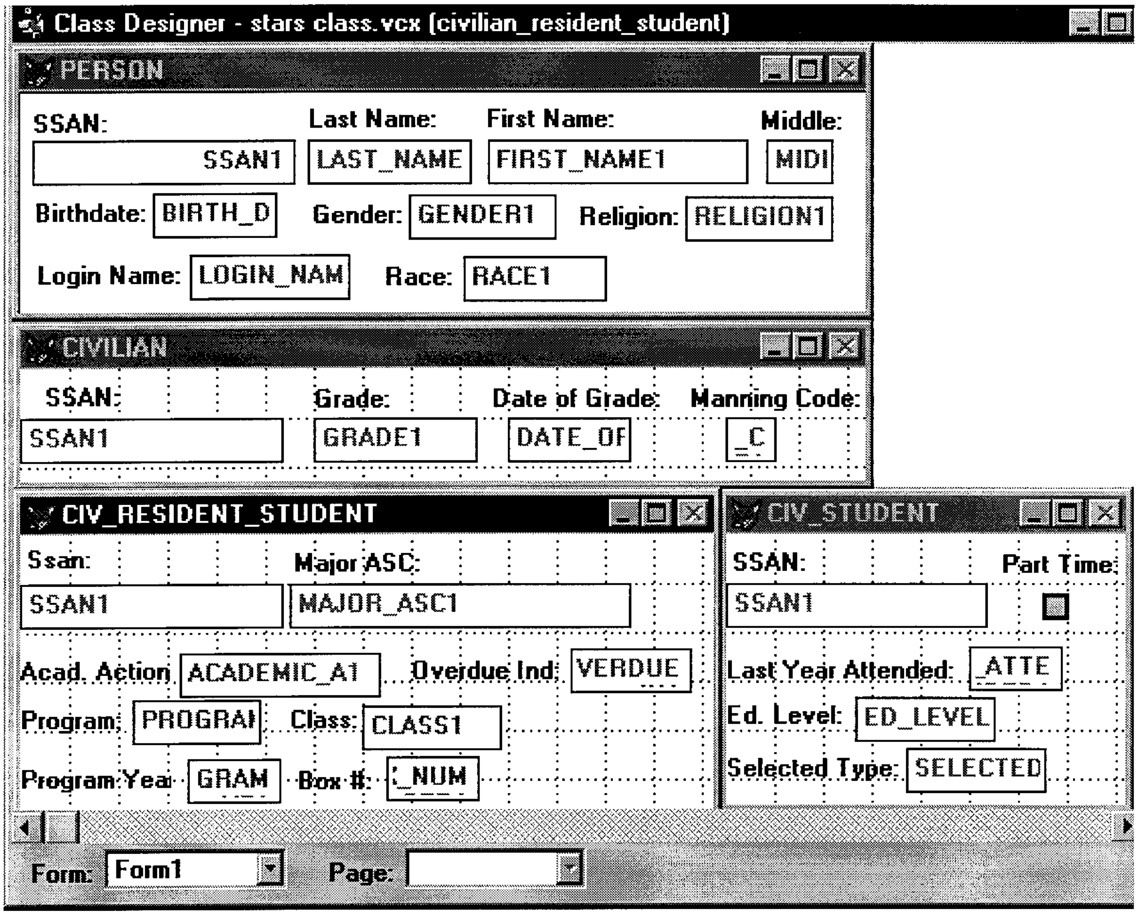


Figure 41: Civilian_resident_student's Class

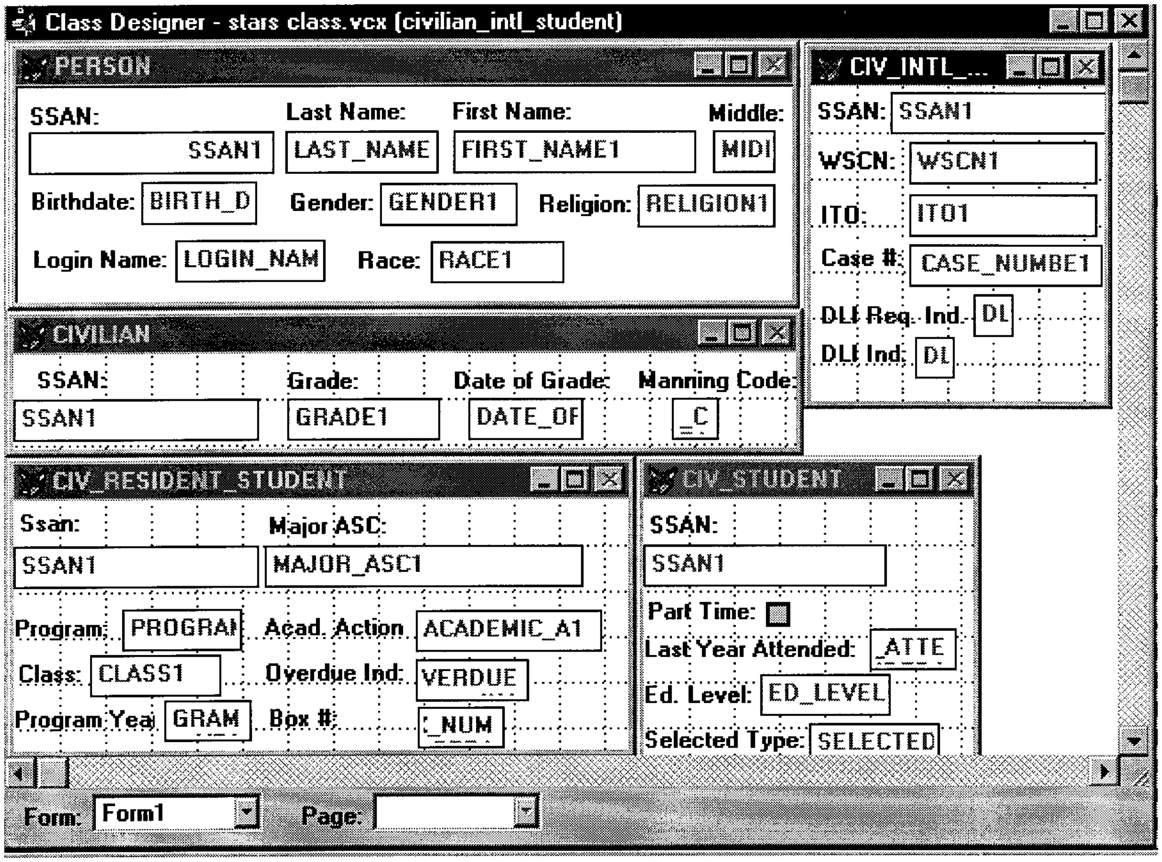


Figure 42: Civilian_INTL_student's Class

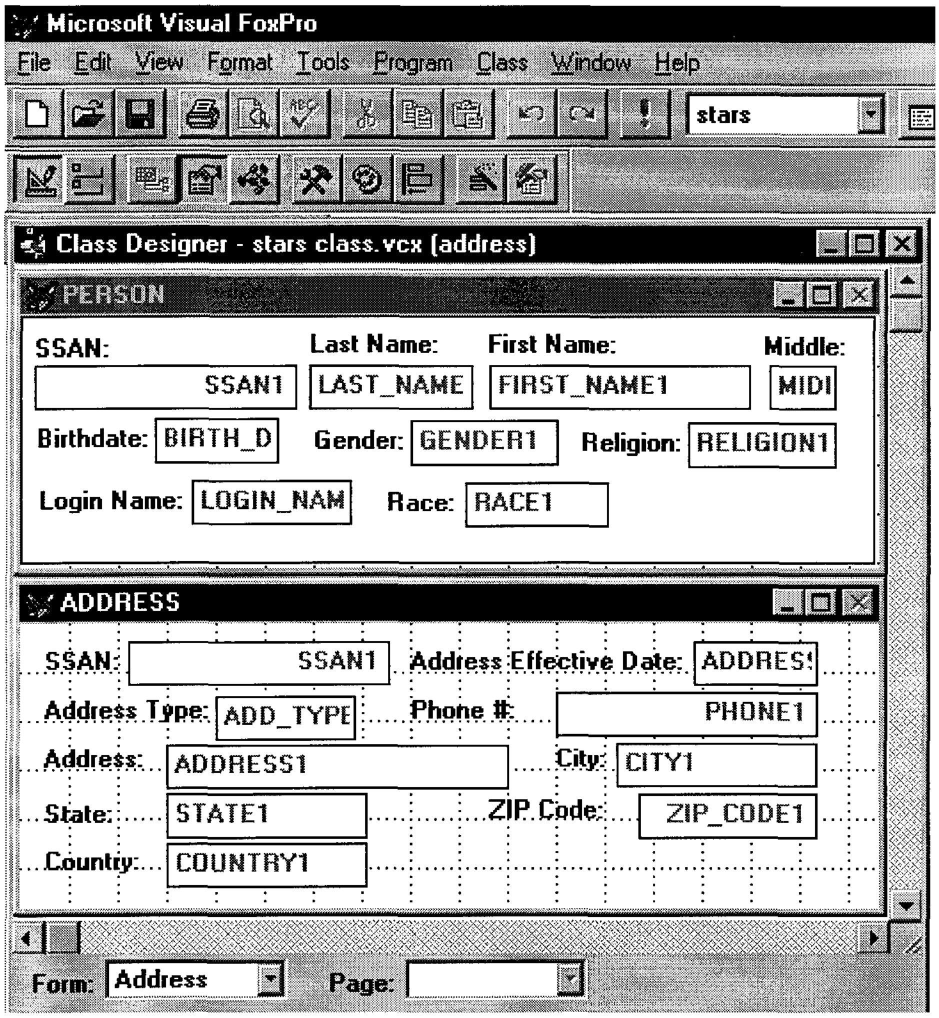


Figure 43: Address' Class

Appendix F: List of Abbreviations

AFIT - Air Force Institute of Technology

AFITSIS - Air Force Institute of Technology Student Information System

AFIT/SC - Air Force Institute of Technology Communication Computer System

CASE - Computer-Aided Software Engineering

DBA - Database Administrator

DBMS - Database Management System

DFD - Data Flow Diagram

ER - Entity Relationship

OODBMS - Object-Oriented Database Management System

RDB - Relational Database

RDBMS - Relational Database Management System

SQL - Structured Query Language

STARS - Student Tracking and Registration System

Bibliography

1 Blaha, Michel and William J. Premerlani. An Approach for Reverse Engineering of Relational Databases. Communications of the ACM. May 1994. Vol. 37, No. 5.

2 Blaha, Michel and William J. Premerlani. Observed Idiosyncrasies of Relational Database Designs. IEEE Software, 1995, pp. 116-125.

3 STARS User's Manual (AFIT Database). System Research Laboratories, 1987.

4 Cerney, Barbara, Capt. Sullivan David, and Kathleen Hale. AFIT/SCQ Personal Interview, May 1995.

5 Chikofsky, Elliot, James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, January 1990, pp. 14-15.

6 Sneed, Harry M. Planning the Reengineering of Legacy Systems. IEEE Software, January 1995, pp. 25-26.

7 Hainaut, J.L., and others. Contribution to a Theory of Database Reverse Engineering. International Conference on Software Engineering, Workshop on Reverse Engineering, May 1993, Baltimore, Maryland.

8 Jacobson, Ivar and Fredrik Lindstrom. Re-engineering of Old Systems to an Object-oriented Architecture. Proceedings, OOPSLA 1991, pp. 340-350.

9 Bennett, Keith. Legacy Systems: Coping With Success. IEEE Software, April 1995, pp. 19-23.

10 Burleson, Donald. Practical Application of Object-Oriented Techniques to Relational Databases. Wiley-QED Publication, 1994.

11 Rumbaugh, James and others. Object-Oriented Modeling and Design. Prentice-Hall, 1991.

12 ERwin User's Guide. Logic Works, Inc, 1995.

13 Microsoft Visual Foxpro Developer's Guide. Microsoft Corporation, 1995.

Vita

Major Pedro Arthur Linhares Lima [redacted] He entered the Air Force Preparatory School of Cadets (EPCAR) in Barbacena, Minas Gerais in 1975, and attended the Brazilian Air Force Academy, where he was graduated in December of 1981. He entered the Catholic University of Rio de Janeiro (PUC-RJ), where was awarded the degree of Bachelor in Systems Analysis in June of 1984. His first assignment was at the Air Force Computer Center of Rio de Janeiro (CCA-RJ), where he worked as a systems analyst for the Flight's Statistics System Project. In January of 1989 he was assigned to Staff and War College (ECEMAR), where he worked as a systems analyst for the War Games Project. In January of 1992 he was assigned to Air Force Computer Science and Statistics Department (DIRINFE). In June of 1994 Major Pedro Lima entered the Air Force Institute of Technology as a Master candidate in computer science.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1996	Master's Thesis

4. TITLE AND SUBTITLE
A Methodology for Reengineering Relational Databases to an Object-Oriented Database

5. FUNDING NUMBERS

6. AUTHOR(S)
Pedro Arthur Linhares Lima

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/96J-01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFIT/SC
2950 P. Street
WPAFB, OH 45433-7765

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

This research proposes and evaluates a methodology for reengineering a relational database to an object-oriented database. We applied this methodology to reengineering the Air Force Institute of Technology Student Information System (AFITSIS) as our test case. With this test case, we could verify the applicability of the proposed methodology, especially because AFITSIS comes from an old version of Oracle RDBMS. We had the opportunity to implement part of the object model using an object-oriented database, and we present some peculiarities encountered during this implementation. The most important result of this research is that it demonstrated that the proposed methodology can be used for reengineering an arbitrarily selected relational database to an object-oriented database. It appears that this approach can be applied to any relational database.

14. SUBJECT TERMS
AFITSIS, reengineering, reverse engineering, information system design, OODBMS design, database OMT

15. NUMBER OF PAGES
103

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL